

937

Deitel 编程金典

TP312C

D43a

# C++ 编程金典

(第3版)

[美] H. M. Deitel, P. J. Deitel 著

周 靖 黄都培 译

杨小平 审校



A1026813

清华大学出版社

## 作者简介

**H. M. Deitel 博士**: Deitel & Associates 公司首席执行官,在计算机领域已有 40 年的工作经验,无论专业技术还是学校教育,均有非常高的造诣。他是全球知名的计算机科学导师以及培训班专业讲师。Deitel 博士拥有麻省理工学院的学士和硕士学位,以及波士顿大学的哲学博士学位。他参与过 IBM 和 MIT 的一系列领先于时代的虚拟内存操作系统项目,研究成果如今已广泛应用于 UNIX、Windows NT、OS/2 和 Linux 系统中。他从事大学教育 20 余年,与其子 P. J. Deitel 成立 Deitel & Associates 公司之前,一直担任波士顿大学计算机科学系主任。他创作及与他人合著了数十本书,并参与了多媒体产品的开发。更让人佩服的是,老当益壮的他,现在的出书量居然有增无减。多年来,他的作品已被翻译成简体中文、繁体中文、日语、俄语、西班牙语、朝鲜语、法语、波兰语以及葡萄牙语等,畅销全球。

**P. J. Deitel**: Deitel & Associates 公司执行副总裁,毕业于麻省理工学院的斯隆工商管理学院,主修信息技术。在 Deitel & Associates 公司,他负责向业内许多知名客户讲授 Java、C、C++、因特网/万维网课程,他们的客户有康柏、朗讯、Sun、NASA 肯尼迪航空中心、美国国家大风暴实验室(NSSL)、IBM 以及其他许多公司和机构。他负责为计算机机构联盟波士顿分部讲授 C++ 和 Java 课程。他目前正利用 Deitel & Associates、Prentice Hall 以及“美国技术教育网络”联合投资的一笔风险基金,利用卫星技术提供技术培训与课程。

Deitel 父子二人,先后参与了多本入门级大学计算机编程语言教材的编写工作,所有这些教材都非常畅销。

# 目 录

## 前言

## 第 1 章 计算机和C++ 编程概述 ..... 1

- 1.1 简介 ..... 1
- 1.2 计算机是什么 ..... 3
- 1.3 计算机的构成 ..... 3
- 1.4 操作系统的进化 ..... 4
- 1.5 个人计算、分布式计算和客户机/服务器计算 ..... 5
- 1.6 机器语言、汇编语言和高级语言 ..... 5
- 1.7 C 和C++ 发展简史 ..... 7
- 1.8 C++ 标准库 ..... 8
- 1.9 Java 和《Java 程序设计》 ..... 8
- 1.10 其他高级语言 ..... 9
- 1.11 结构化编程 ..... 9
- 1.12 关键的软件趋势:对象技术 ..... 10
- 1.13 典型C++ 环境的基础 ..... 12
- 1.14 硬件发展趋势 ..... 14
- 1.15 因特网发展简史 ..... 14
- 1.16 万维网发展简史 ..... 16
- 1.17 C++ 和本书的常规注意事项 ..... 16
- 1.18 C++ 编程简述 ..... 17
- 1.19 一个简单的程序:打印一行文字 ..... 17
- 1.20 另一个简单的程序:两个整数相加 ..... 21
- 1.21 内存的概念 ..... 24
- 1.22 算术运算 ..... 25
- 1.23 判断:相等性和关系操作符 ..... 28
- 1.24 对象思想:对象技术及 UML 简介 ..... 32
- 1.25 小结 ..... 36

## 第 2 章 控制结构 ..... 52

- 2.1 简介 ..... 52
- 2.2 算法 ..... 52
- 2.3 伪代码 ..... 53
- 2.4 控制结构 ..... 53
- 2.5 if 选择结构 ..... 56
- 2.6 if/else 选择结构 ..... 57

- 2.7 while 重复结构 ..... 61
- 2.8 算法设计:案例分析 1(计数器控制重复) ..... 62
- 2.9 算法设计:案例分析 2(标记控制重复) ..... 65
- 2.10 算法设计:案例分析 3(嵌套控制结构) ..... 71
- 2.11 赋值操作符 ..... 75
- 2.12 自增和自减操作符 ..... 76
- 2.13 计数器控制重复的本质 ..... 79
- 2.14 for 重复结构 ..... 80
- 2.15 for 结构用法示例 ..... 84
- 2.16 switch 多选结构 ..... 88
- 2.17 do/while 重复结构 ..... 93
- 2.18 break 和 continue 语句 ..... 95
- 2.19 逻辑操作符 ..... 96
- 2.20 混淆相等性操作符(==)和赋值操作符(=) ..... 99
- 2.21 结构化编程小结 ..... 100
- 2.22 【可选案例分析】对象思想:标识问题所牵涉的类 ..... 105
- 2.23 小结 ..... 116

## 第 3 章 函数 ..... 142

- 3.1 简介 ..... 142
- 3.2 C++ 中的程序组件 ..... 142
- 3.3 数学库函数 ..... 143
- 3.4 函数 ..... 144
- 3.5 函数定义 ..... 145
- 3.6 函数原型 ..... 148
- 3.7 头文件 ..... 150
- 3.8 生成随机数 ..... 151
- 3.9 示例:博彩游戏和 enum 简介 ..... 156
- 3.10 存储类 ..... 159
- 3.11 作用域规则 ..... 161
- 3.12 递归 ..... 164
- 3.13 递归应用示例:费波拉奇数列 ..... 167

3.14	递归和迭代的对比 .....	170	6.1	简介 .....	349
3.15	使用空参数列表的函数 .....	172	6.2	结构定义 .....	350
3.16	内联函数 .....	173	6.3	访问结构成员 .....	350
3.17	引用和引用参数 .....	174	6.4	用 struct 实现用户自定义类型 Time .....	351
3.18	默认实参 .....	178	6.5	用 class 实现 Time 抽象数据类型 .....	353
3.19	一元作用域分辨符 .....	179	6.6	类作用域和访问类成员 .....	358
3.20	函数重载 .....	180	6.7	接口同实现方法的分离 .....	360
3.21	函数模板 .....	182	6.8	控制对成员的访问 .....	363
3.22	【可选案例分析】对象思想:标识类 的属性 .....	184	6.9	访问函数和工具函数 .....	366
3.23	小结 .....	189	6.10	初始化类对象:构造函数 .....	369
第4章	数组 .....	215	6.11	在构造函数中使用默认参数 .....	369
4.1	简介 .....	215	6.12	使用析构函数 .....	373
4.2	数组 .....	215	6.13	何时调用构造函数和析构函数 .....	373
4.3	声明数组 .....	217	6.14	使用数据成员和成员函数 .....	376
4.4	数组用法示例 .....	217	6.15	微妙的陷阱:返回对 private 数据成 员的引用 .....	380
4.5	将数组传给函数 .....	231	6.16	通过默认的按位成员复制赋值 .....	383
4.6	数组排序 .....	235	6.17	软件重用性 .....	384
4.7	案例分析:利用数组计算均数、中 位数和众数 .....	237	6.18	【可选案例分析】对象思想:编写 电梯模拟程序所需的类 .....	385
4.8	搜索数组:线性搜索和二元搜索 .....	241	6.19	小结 .....	395
4.9	多下标数组 .....	246	第7章	类和数据抽象(二) .....	405
4.10	【可选案例分析】对象思想:标识类 的行为 .....	252	7.1	简介 .....	405
4.11	小结 .....	258	7.2	常量对象和常量成员函数 .....	405
第5章	指针和字符串 .....	276	7.3	合成:对象作为类成员 .....	412
5.1	简介 .....	276	7.4	友元函数和友元类 .....	418
5.2	指针变量声明和初始化 .....	276	7.5	使用 this 指针 .....	421
5.3	指针操作符 .....	277	7.6	用 new 和 delete 实现动态内存分配 .....	425
5.4	按引用调用函数 .....	280	7.7	静态类成员 .....	426
5.5	使用带指针的 const 限定符 .....	283	7.8	数据抽象和信息隐藏 .....	431
5.6	使用引用调用的冒泡排序 .....	289	7.9	容器类和迭代器 .....	434
5.7	指针表达式和指针算法 .....	294	7.10	代理类 .....	434
5.8	指针和数组的关系 .....	296	7.11	【可选案例分析】对象思想:为电梯 模拟程序中的类编写程序 .....	436
5.9	指针数组 .....	299	7.12	小结 .....	462
5.10	案例分析:洗牌和发牌模拟程序 .....	300	第8章	操作符重载 .....	469
5.11	函数指针 .....	304	8.1	简介 .....	469
5.12	字符和字符串处理概述 .....	309	8.2	操作符重载的基础知识 .....	469
5.13	【可选案例分析】对象思想:对象间 的合作 .....	317	8.3	操作符重载的限制条件 .....	470
5.14	小结 .....	322			
第6章	类和数据抽象(一) .....	349			



8.4	类成员操作符函数与友元函数操作符函数的对比 .....	472	10.8	虚拟析构函数 .....	574
8.5	重载流插入与流读取操作符 .....	473	10.9	案例分析:继承接口和实现 .....	574
8.6	重载一元操作符 .....	475	10.10	多态性、虚拟函数和动态绑定的本质 .....	582
8.7	重载二元操作符 .....	476	10.11	小结 .....	585
8.8	案例分析:Array 类 .....	477	<b>第 11 章 C++ 输入/输出流</b> .....	<b>589</b>	
8.9	类型转换 .....	488	11.1	简介 .....	589
8.10	案例分析:String 类 .....	489	11.2	流 .....	589
8.11	重载 ++ 和 -- .....	499	11.3	输出流 .....	592
8.12	案例分析:Date 类 .....	500	11.4	输入流 .....	595
8.13	小结 .....	505	11.5	成员函数 read, gcount 和 write 的无格式输入/输出 .....	601
<b>第 9 章 继承</b> .....	<b>517</b>		11.6	流操纵元 .....	602
9.1	简介 .....	517	11.7	流格式状态 .....	607
9.2	继承:基类与派生类 .....	518	11.8	流错误状态 .....	616
9.3	protected 成员 .....	520	11.9	把输出流连接到输入流 .....	618
9.4	基类指针向派生类指针的强制类型转换 .....	520	11.10	小结 .....	618
9.5	使用成员函数 .....	525	<b>第 12 章 模板</b> .....	<b>630</b>	
9.6	在派生类中改写基类成员 .....	526	12.1	简介 .....	630
9.7	public、protected 和 private 继承 .....	529	12.2	函数模板 .....	630
9.8	直接基类和间接基类 .....	530	12.3	重载模板函数 .....	633
9.9	在派生类中使用构造和析构函数 .....	530	12.4	类模板 .....	634
9.10	派生类向基类的隐式转换 .....	534	12.5	类模板与无类型参数 .....	639
9.11	继承在软件工程中的应用 .....	534	12.6	模板与继承 .....	639
9.12	合成与继承 .....	536	12.7	模板与友元 .....	640
9.13	“使用”关系和“知道”关系 .....	536	12.8	模板与静态数据成员 .....	641
9.14	案例分析:Point、Circle 和 Cylinder 类 .....	536	12.9	小结 .....	641
9.15	多重继承 .....	544	<b>第 13 章 异常处理</b> .....	<b>646</b>	
9.16	【可选案例分析】对象思想:在电梯模拟程序中集成继承 .....	548	13.1	简介 .....	646
9.17	小结 .....	554	13.2	何时使用异常处理 .....	648
<b>第 10 章 虚拟函数和多态性</b> .....	<b>560</b>		13.3	其他错误处理方法 .....	648
10.1	简介 .....	560	13.4	C++ 异常处理基础:try、throw 和 catch .....	649
10.2	类型域和 switch 语句 .....	560	13.5	简单的异常处理例子:除数为 0 .....	649
10.3	虚拟函数 .....	560	13.6	抛出异常 .....	651
10.4	抽象基类和具体类 .....	561	13.7	捕捉异常 .....	652
10.5	多态性 .....	562	13.8	重抛出异常 .....	655
10.6	案例分析:使用多态性的工资发放系统 .....	564	13.9	异常的规约 .....	656
10.7	新类和动态绑定 .....	574	13.10	处理意外异常 .....	657
			13.11	堆栈解退 .....	657
			13.12	构造函数、析构函数与异常处理 .....	658

13.13	异常与继承 .....	659	16.10	字符串转换函数 .....	779
13.14	处理 new 失败 .....	659	16.11	字符串处理函数库的查找函数 .....	783
13.15	auto_ptr 类与动态内存分配 .....	663	16.12	字符串处理函数库中的内存处理 函数 .....	788
13.16	标准库异常的层次结构 .....	665	16.13	字符串处理函数库中的其他函数 .....	791
13.17	小结 .....	665	16.14	小结 .....	792
<b>第 14 章</b>	<b>文件处理 .....</b>	<b>674</b>	<b>第 17 章</b>	<b>预处理程序 .....</b>	<b>805</b>
14.1	简介 .....	674	17.1	简介 .....	805
14.2	数据的层次结构 .....	674	17.2	预处理程序指令#include .....	805
14.3	文件和流 .....	676	17.3	预处理程序指令#define;符号常量 .....	806
14.4	创建顺序访问文件 .....	677	17.4	预处理程序指令#define;宏指令 .....	806
14.5	读取顺序访问文件中的数据 .....	680	17.5	条件编译 .....	808
14.6	更新顺序访问文件 .....	686	17.6	预处理程序指令#error 与#pragma .....	808
14.7	随机访问文件 .....	686	17.7	操作符#与## .....	809
14.8	建立随机访问文件 .....	687	17.8	行号 .....	809
14.9	向随机访问文件随机写入数据 .....	689	17.9	预定义符号常量 .....	809
14.10	从随机访问文件中顺序读取数据 .....	691	17.10	宏指令(assert) .....	810
14.11	案例分析:事务处理程序 .....	693	17.11	小结 .....	810
14.12	对象的输入/输出 .....	699	<b>第 18 章</b>	<b>C 遗留代码 .....</b>	<b>815</b>
14.13	小结 .....	699	18.1	简介 .....	815
<b>第 15 章</b>	<b>数据结构 .....</b>	<b>707</b>	18.2	UNIX 与 DOS 系统中的重定向 输入/输出 .....	815
15.1	简介 .....	707	18.3	变长参数列表 .....	816
15.2	自引用类 .....	708	18.4	使用命令行参数 .....	818
15.3	动态内存分配 .....	708	18.5	编译多个源文件程序的相关说明 .....	819
15.4	链表 .....	709	18.6	用函数 exit 与 atexit 终止程序运行 .....	821
15.5	堆栈 .....	721	18.7	类型限定符 volatile .....	822
15.6	队列 .....	725	18.8	整数和浮点数常量的后缀 .....	822
15.7	树 .....	728	18.9	信号处理 .....	823
15.8	小结 .....	735	18.10	用 calloc 与 realloc 动态内存分配 .....	825
<b>第 16 章</b>	<b>位、字符、字符串与结构 .....</b>	<b>757</b>	18.11	无条件转向语句;goto .....	825
16.1	简介 .....	757	18.12	联合体 .....	826
16.2	结构的定义 .....	757	18.13	接合规约 .....	829
16.3	结构的初始化 .....	759	18.14	小结 .....	830
16.4	在函数中使用结构 .....	759	<b>第 19 章</b>	<b>string 类和字符串流处理 .....</b>	<b>837</b>
16.5	关键字 typedef .....	759			
16.6	示例:高性能洗牌与发牌模拟程序 .....	760			
16.7	位操作符 .....	762			
16.8	位段 .....	770			
16.9	字符处理函数库 .....	773			

19.1	简介 .....	837	21.8	操作符关键字 .....	960
19.2	字符串的赋值与拼接 .....	838	21.9	显式构造函数 .....	962
19.3	比较字符串 .....	840	21.10	mutable 类成员 .....	967
19.4	子串 .....	843	21.11	类成员指针(, * 和 -> *) .....	968
19.5	交换字符串 .....	843	21.12	多重继承和虚拟基类 .....	970
19.6	字符串的特性 .....	844	21.13	结束语 .....	974
19.7	查找字符串中的字符 .....	846	21.14	小结 .....	974
19.8	替换字符串中的字符 .....	848	附录 A	操作符的优先级和结合性 .....	980
19.9	在字符串中插入字符 .....	850	附录 B	ASCII 字符集 .....	982
19.10	转换为 C 风格的 char * 字符串 ...	851	附录 C	数值系统 .....	983
19.11	迭代器 .....	853	C.1	简介 .....	983
19.12	字符串流处理 .....	854	C.2	将二进制数简化为八进制和十 六进制数 .....	985
19.13	小结 .....	858	C.3	将八进制和十六进制数转换为 二进制数 .....	986
第 20 章	标准模板库 (STL) .....	864	C.4	将二进制、八进制或十六进制 转换为十进制 .....	987
20.1	标准模板库 STL 简介 .....	864	C.5	将十进制转换为二进制、八进 制或十六进制 .....	987
20.2	序列容器 .....	874	C.6	负的二进制数:2 的补值记号法 ...	989
20.3	关联容器 .....	886	C.7	小结 .....	990
20.4	容器适配器 .....	894	附录 D	因特网和万维网上的 C++ 资源 .....	994
20.5	算法 .....	899	D.1	资源 .....	994
20.6	bitset 类 .....	930	D.2	教程 .....	995
20.7	函数对象 .....	933	D.3	FAQ .....	995
20.8	小结 .....	936	D.4	Visual C++ .....	996
第 21 章	标准 C++ 语言的增补 .....	947	D.5	comp. lang. C++ .....	996
21.1	简介 .....	947	D.6	编译工具 .....	998
21.2	布尔数据类型 .....	947	D.7	开发工具 .....	999
21.3	static _ cast 操作符 .....	949	D.8	标准模板库 .....	999
21.4	const _ cast 操作符 .....	951			
21.5	reinterpret _ cast 操作符 .....	952			
21.6	名称空间 .....	953			
21.7	运行时类型信息 (RTTI) .....	956			

# 第1章 计算机和C++ 编程概述

## 学习目标

- 理解计算科学的基本概念
- 熟悉不同类型的编程语言
- 理解一个典型的C++ 程序开发环境
- 会用C++ 写一个简单的计算机程序
- 会使用简单的输入和输出语句
- 熟悉基本数据类型
- 会使用算术操作符
- 理解算述操作符的优先级
- 会写简单的用于做出决定的语句

## 1.1 简介

欢迎进入C++ 的世界！我们经过艰苦努力，为大家带来一本具有丰富内涵、能真正寓教于乐的计算机参考书。C++ 是一种困难的语言，一般只适合那些有经验的程序员。但正由于此，本书才能在琳琅满目的C++ 参考书中鹤立鸡群：

- 它适合无论有无编程经验，只要有志于从事技术的人
- 它适合有经验的程序员，他们想从更深的层次来看待这种语言

有人也许会问，同一本书怎么可能同时适用于高低两个层次的读者呢？其中的关键在于，我们以成熟的“结构化编程”及“基于对象的编程”技术为准，从头至尾始终强调如何编写“思路清晰”的程序。对于非程序员出身的人来说，利用本书可少走弯路。我们写作时采取一种清晰的、平铺直叙的方式，其间有大量插图作为衬托。另外更重要的是，本书提供了数百个功能完整的C++ 程序，同时展示了这些程序实际运行时得到的正确输出结果。这便是我们独有的“活代码”（Live - Code）方式。本书每个示例程序都可从我们的网站（[www.deitel.com](http://www.deitel.com)）下载。

前5章介绍了计算机、计算机编程和C++ 计算机程序设计语言的基础。一些上过我们课的新生告诉我们，第1~5章的内容为他们在以后的章节里深入理解C++ 打下了坚实的基础。有经验的程序员往往只是快速翻阅一下前5章，但从本书剩下的部分开始C++ 的学习时，却发现自己还存在不少问题，不得不回头重新学习基础知识。

许多高级程序员告诉我们，他们非常喜欢这种强调结构化编程的写作手法。他们通常喜欢用Pascal 或C 进行结构化编程。但是，由于以前并没有系统学习过结构化编程的理论知识，所以写出来的往往不是最优化的代码。通过学习本书前几章的结构化编程后，他们有

效地改进其 C 和 Pascal 编程风格。因此,无论是新手,还是有经验的程序员,都会欣赏本书信息丰富、有趣而富有挑战性的特色。

许多人对计算机的强大功能都较为了解。利用本书,大家会学习到如何指挥自己的机器,真正完成实际工作。无论如何,计算机(通常叫硬件)的控制都必须通过软件(亦即一系列用来指挥计算机采取行动以及做出选择的指令)进行。C++ 是当今最流行的软件开发语言之一。本书概述了 C++ 的编程理论。所用的 C++ 版本在美国境内通过美国标准化协会(ANSI)得到了标准化;在世界范围内,则通过国际标准化组织(ISO)的努力得到了标准化。

如今,各个领域内的人几乎越来越离不开计算机。在其他成本都在稳步提升的同时,计算方面的成本却呈显著下降态势——这完全应归功于硬件和软件技术的快速发展。25~30 年前,需要数个大房间才能摆下的大型计算机,以及那些动辄数百万美元的“超级芯片”(现在只需指甲大小的硅芯片)成本也降到了每片几美元左右。不过有意思的是,“硅”是我们这个地球上不值钱的东西之一。在海边随手抓一把沙子,里面含的绝大多数元素便是“硅”。硅芯片技术的问世,使得计算机成本变得异常低廉,也直接促成了如今计算机在各行各业的快速普及。在商业、工业、政府以及我们的个人生活方面,计算机都能提供强有力的帮助。

本书之所以具有一定的挑战性,有几方面的原因。过去几年,你的同事可能学习了 C 或 Pascal(此为他们学习的第一种编程语言)。而现在,你实际需要同时学习 C 和 C++! 为什么? 因为 C++ 本身已包括 C,并在其基础上添加了许多新东西。

你的同事可能学过一种名为结构化编程的编程方法。而你既要学习结构化编程,也要学习一种更令人激动的新方法,名为面向对象编程。那么,我们为什么要兼顾两者呢? 因为对于下一个 10 年来说,“面向对象”无疑是最关键的编程方法。在本书的学习过程中,你会创建和使用各种各样的对象。但你往往会发现,那些对象的内部结构最好是用结构化编程技术来构建。此外,对象的处理逻辑有时也最好通过结构化编程来表达。

同时讲授两种方法的另一个原因是,目前有大量、基于 C 的系统需要迁移至基于 C++ 的系统。有为数不少的所谓“C 遗留代码”仍在大量使用中。在近 25 年时间里,C 得到了广泛的应用,而且它在最近这几年的应用也早显著提升的态势。一旦人们学会了 C++,便会发现其强于 C 的地方,所以通常会选择全面转向 C++ 系统。但在这之前,首先需要将传统系统转换成 C++。然后开始采用一系列所谓“C++ 对 C 的增强”特性,改善自己编写“类似 C”程序的写作风格。最后才开始全面利用 C++ 的面向对象编程能力,最终发挥出这种语言的全部潜力。

对于编程语言,一个有趣的现象是大多数厂商推出的都是 C/C++ 合成产品,而不是独立的产品。这样一来,用户可暂时沿用 C 进行编程;时机成熟时再逐渐转移到 C++。

目前,C++ 已成为构建高性能计算机系统的首选语言。但针对本书面向的目标读者,不能在他们的第一门编程课程中便开始讲授 C++ 呢? 答案是肯定的。9 年前,我们也面临过类似的挑战;那时,在学生们的第一门计算机科学课程中,Pascal 是首选的入门语言。当时,我们编著了《C 程序设计》。现在,全世界数百所大学都在使用《C 程序设计(第三版)》。事实证明,围绕本书展开的课程与以前基于 Pascal 的课程同样有效。两者没有明显的差异,但学生们拥有了更强烈的学习动机,因为他们知道在自己以后的高级课程以及职业生涯中,会

更多地使用 C,而不再是 Pascal。学习 C 的学生还知道自己将更好地为C++ 的学习做好准备,同时更有信心迎接面向因特网的、基于C++ 的新型语言(即 Java)的挑战。

在本书的前 5 章,将介绍C++ 中的结构化编程概念。此为C++ 的“C 部分”,亦即“C++ 对 C 的增强”。在本书其他部分,将介绍C++ 中的面向对象编程概念。但我们并不希望你从第 6 章才开始学习面向对象的编程。因此,在前 5 章每一章结束的时候,都提供了一个名为“对象思想”的小节。这些小节介绍了面向对象编程的基本概念和术语。这样一来,到第 6 章“类和数据抽象”时,你便已做好了充分的准备,可直接开始用C++ 创建对象,以及着手编写面向对象的程序。

第 1 章首先描述计算机和计算机编程的基础知识,接着马上介绍如何编写简单的C++ 程序。最后将帮助你开始进行“对象思想”的练习。

从现在起,大家将开始一段美妙的、令人激动的、充满挑战的,但又令人回味无穷的学习之旅。在这个旅程中,如果碰到问题,不妨发信给:deitel@deitel.com 或浏览我们的网站:www.deitel.com,所有问题都会很快得到回复。我们希望你在使用《C++ 编程金典(第 3 版)》的过程中,保持愉快的心情。

## 1.2 计算机是什么

所谓计算机,是能执行计算并作出逻辑决策的一种设备,其运算速度比人脑快数百万乃至数十亿倍。例如,现在许多个人计算机每秒都能执行数亿次加法运算。换作人,即使有一个计算器的帮助,也需数十年才能完成同等数量的运算,而功能强劲的个人计算机一秒钟便能完成(如果你是一位喜欢动脑筋的人,这里还有一些提示:如何判断一个人是否将数字正确地加到一起?又如何判断一部计算机将数字正确地加到一起?)今天最快的超级计算机每秒可执行数千亿次加法运算——数十万人连续工作一年才能完成所有这些运算!此外,每秒能执行万亿条指令的计算机已在实验室中研制成功!

计算机处理数据时,需在一系列指令的控制下进行,这一系列指令便统称计算机程序。程序引导计算机一步步完成预先规定好的操作。那么,是谁规定了这些操作呢?正是我们这些计算机程序员!

计算机由多种设备构成,比如键盘、屏幕、鼠标、磁盘、内存、光盘驱动器以及处理单元等等。我们将这些设备统称为硬件。在计算机上运行的计算机程序称为软件。近年来,硬件价格已显著降低,目前的个人计算机形同普通家电。但令人遗憾的是,软件开发的行情依然低迷,成本越来越高。为什么呢?这是由于软件开发的技术并没有显著改进;另一方面则是程序员必须开发出功能更强、更复杂的应用程序。在本书中,大家将会学到一系列屡经考验和修订、目前已非常成熟的软件开发技术。利用这些开发技术,软件开发的成本可显著降低。其中包括:结构化编程、自上而下求精法、功能化、基于对象的编程、面向对象的编程以及由事件驱动的编程等。

## 1.3 计算机的构成

尽管物理外观有所差别,但几乎所有计算机都由 6 大逻辑单元或区域构成,它们是:

(1) 输入单元。这是计算机的“接收”区域。可从各种输入设备那里获取信息(数据和计算机程序),并将这些信息交由其他单元进行处理。今天,大多数信息都是用键盘和鼠标输入到计算机里的。其他输入设备还包括麦克风,可将你说的话输入计算机;扫描仪,可扫描图片;以及数码相机/摄像机,可拍摄照片或影片。

(2) 输出单元。这是计算机的“发送”区域。它可取得计算机处理过的信息,并把它放在各种输出设备上,以便在计算机的外面使用这些信息。如今计算机输出的大多数内容都直接显示在屏幕上,或打印在纸上,或用于对其他设备进行控制。

(3) 存储单元。这是计算机进行快速存取、但容量并不大的一个“仓库”单元,可供临时性存储数据。它会保留通过输入单元输入的信息。需要时,便可马上取用并处理之。内存单元还可放置处理好的信息,直至这些信息由输出设备传到输出设备上。在此,我们通常将内存单元叫作内存或主存存储器。

(4) 算术和逻辑单元(ALU)。这是计算机负责“生产”的单元。它可执行像加、减、乘、除这样的计算。另外,ALU 里还包含了一些决策(选择)机制。例如,利用这种机制,计算机可在内存单元中比较两个项目,以判断它们是否相等。

(5) 中央处理单元(CPU)。这是计算机的“管理”单元。它担当着计算机的“协调人”的角色,负责监督其他单元的操作。需要将信息读入内存单元时,CPU 会向输入单元发出指示;如计算时需使用来自内存单元的信息,则会向 ALU 发出指示;最后,若需将来自内存单元的信息发给特定的输出设备,就会向输出单元发出指示。

(6) 辅助存储单元。这是计算机长期性的、大容量的“仓储”单元。一些不常被其他单元用到的程序或数据通常放在辅助存储设备里(比如磁盘),直至再次需要它们——这便是几个小时、几个月乃至数年之后的事情了。对放在辅助存储单元中的信息来说,访问它们的速度通常要慢于访问主内存中的数据。不过另一方面,辅助存储单位成本要比主存存储器的成本低得多。

## 1.4 操作系统的进化

早期计算机每次只能执行一项作业或任务。如计算机以这种方式工作,则通常叫做单用户批处理系统。在这种系统中,计算机一次执行一个程序,同时以组或成批的方式来处理数据。在这种早期的系统中,用户通常先将自己的工作制作成一叠叠的穿孔卡,再交由计算机中心处理。要想获得输出结果,一般要等待数小时乃至数天。

为此,人们开发出了一种软件系统,以便更容易地使用计算机。这种软件系统便叫作操作系统。早期的操作系统可自行在不同的作业之间切换,免去计算机操作员亲自切换的麻烦,从而提高了工作效率。同时,也增大了计算机能处理的工作量(专业术语叫吞吐量)。

随着计算机越来越强劲,人们也意识到单用户批处理方式在利用计算机资源的时候,效率仍然十分低下,这是由于大多数时间都浪费在等待慢吞吞的输入/输出设备工作。于是,人们想到,多个工作或任务为什么不能共享计算机资源,进一步提高利用率呢?于是,多道程序(Multiprogramming)的概念问世了,它涉及到计算机内多项工作的同时——计算机在不同的作业之间共享自己的资源。不过在早期的多路程序操作系统上,用户仍需使用穿

孔卡在控制台上提交作业,等几个小时或者几天才能拿到结果。

20 世纪 60 年代,活跃于计算机行业以及大学校园的几个开发小组先后提出了时间共享操作系统的概念。所谓时间共享,其实是多路程序的一种特殊情况。在这种情况下,用户通过终端来访问计算机——终端通常只配备了键盘和显示屏。在一个典型的时间共享计算机系统中,同时可以有数十乃至数百名用户共享计算机。不过,计算机实际并不是同时运行所有这些用户的程序。相反,它只运行某个用户任务的一小部分,再转到下一名用户的任务。但是,由于计算机做这种事情的速度非常快,所以每秒钟往往能多次为同一名用户提供服务。这样一来,至少从表面上看,用户们的程序是同时运行的。时间共享的一个优点在于,用户发出请求后,几乎马上可获得响应,不必像以前的计算模式那样,很久才能得到输出结果。

## 1.5 个人计算、分布式计算和客户机/服务器计算

1977 年,苹果计算机公司率先引入了个人计算机的概念。不过当初,由于价格方面的因素,只有发烧友才会对此感兴趣。但没过多久,计算机就变得非常便宜,普通人都可以买回来在家里或公司里使用。1981 年,全球最大的计算机制造商 IBM 公司发布了 IBM 个人计算机。几乎就在一夜之间,个人计算机就成为商业、工业和政府机构的“标准配置”。

不过,早期的这些计算机都属于“单机”设备——人们先在自己的机器上完成工作,再通过软盘传递数据,以实现信息共享。后来,由于这种方式的确不便,人们又想到了“网络”。事实上,尽管早期的个人计算机本身功能并不强大,不足以带动数名用户实现时间共享,但仍可连接到一个计算机网络。连接时,要么使用电话线实现远程连网,要么采用局域网(LAN),在单位内部组建一个网络。分布式计算的概念应运而生。采用分布式计算,单位内的计算工作不必全部集中在一台中央计算机上完成,而是分散到整个网络,由所有连网的计算机协助完成。目前的个人计算机已具有足够强大的计算能力,完全能满足单个用户的计算需求,还能轻松自如地完成基本的通信任务,以电子方式来回传递信息——分布式计算大有潜力可挖!

如今,即使最普通的个人计算机,其运算能力也完全可以和 10 年前需要数百万美元才能买到的大型计算机。比桌面机更高级的是工作站,可让用户体验到更强的运算能力。通过计算机网络,信息可轻松地共享。在网络中一些文件服务器上,可存储一系列通用的程序和数据,以便网内的所有客户机自由取用。这便是所谓的客户机/服务器计算。另外,为各种操作系统、计算机连网以及分布式客户机/服务器应用编写软件时,C 和 C++ 已成为最标准的编程语言之一。对于当今一些流行的操作系统来说,比如 UNIX、Linux 和 Microsoft Windows 系统,都具有本节讨论的各种功能。

## 1.6 机器语言、汇编语言和高级语言

程序员用不同的编程语言编写指令。有的语言是计算机直接能理解的,有的则需经过一系列翻译步骤。如今,全世界有数百种计算机语言正在使用。不过,我们可把它们大致划



分为3类:

- (1) 机器语言;
- (2) 汇编语言;
- (3) 高级语言。

任何计算机都能直接理解自己的机器语言。机器语言是任何一台特定计算机的“自然语言”。这种语言由计算机的硬件设计所定义。在机器语言中,通常包括大量数值串(最终可简化成一系列的1和0),它们指示计算机执行最基本的操作。注意机器语言是依赖于机器的;换句话说,一种机器语言只能在一种特定类型的计算机上使用。不过,对于人而言,机器语言实在令人费解。比如下面这段机器语言代码,其作用是将超时加班工资(OVERPAY)与基本工资(BASEPAY)相加,并将结果存到总工资(GROSSPAY)里

```
+1300042774
+1400593419
+1200274027
```

随着计算机的普及,机器语言便显得太慢,太乏味了,而且还特别容易出错。因此,人们不再使用计算机能直接理解的数字字符串,而是开始采用英语风格的缩写指令,从而对计算机的基本操作进行描述。这些缩写形式便构成了汇编语言的基础。另外,人们开发出了名为汇编程序的一种转换程序,以计算机的超快速度,将汇编语言程序翻译成机器语言。这同汇编语言代码

```
LOAD  BASEPAY
ADD   OVERPAY
STORE GROSSPAY
```

的作用相同,但比机器语言更简洁、更容易理解。

尽管在人看来,这段代码非常容易理解,但除非由汇编程序转换成机器语言,否则计算机还是对此毫无反应——不明白。记住,机器语言是计算机惟一能直接识别的语言!

汇编语言出现后,计算机的应用更加广泛,人们也更能接受它。不过,汇编语言仍然算不上简单语言。即使非常简单的任务,也须编写大量指令。为加快编程速度,又出现了高级语言。以前需要大段汇编指令才可完成的事情,现在只须一条简单语句即可。一种名为编译程序或编译器的转换程序可将高级语言转换为机器语言。使用高级语言,程序员可以采用日常用语编写自己的程序,其中包含了大量常规的数学符号。仍然以前面的例子为例,一个用高级语言写成的工资计算程序只需有代码

```
grossPay = basePay + overTimePay
```

显然,从程序员的角度看,高级语言比机器或汇编语言简单得多。目前,C和C++均是最流行的、功能最强大的高级语言。

不过,将高级语言程序编译成机器语言时,可能耗时较长。所以人们又开发了解释程序,以便直接执行高级语言程序,毋需事先将其编译成机器语言。尽管编译好的程序执行速度比解释过的程序快,但在程序开发环境中,最流行的还是解释程序。在这种环境中,随着新功能的引入,以及错误的不断纠正,需要经常改变程序。一个程序开发好之后,就可生成它的一个编译版本,以便更有效地运行。

## 1.7 C和C++发展简史

C++是在C的基础上演变而来的,而C又是从更老的两种程序语言演变而来的。这两种语言分别是BCPL和B。其中,BCPL由Martin Richards于1967年开发成功,用于编写操作系统软件和编译程序。后来,Ken Thompson以BCPL为基础,在他的B语言中建模构造了许多相似的特性。20世纪70年代,贝尔实验室在一台DEC PCP-7计算机上,用B来创建早期版本的UNIX操作系统。不管BCPL还是B都是“无类型”语言——每个数据项都在内存中占据一个“字”的空间,要由程序员自行决定将一个数据项作为整数,还是作为实数来处理。

C语言由贝尔实验室的Dennis Ritchie在B的基础上研制而成,最初于1972年,在一台DEC PDP-11计算机上实现的。C沿用了BCPL和B的许多重要概念,同时添加了数据类型以及其他一些特性。C最开始以UNIX操作系统的开发语言而著称。如今,大多数操作系统都是用C和(或)C++写成的。C目前可用于大多数计算机,C与硬件无关的。只要设计得当,便有可能写出能移植到其他大多数计算机的C程序。

20世纪70年代末,C已发展为今天的“传统C”、“经典C”或“Kernighan和Ritchie C”。1978年,Prentice Hall出版了Kernighan和Ritchie合著的《The C Programming Language》(C程序语言)一书,C语言从此引起人们的广泛关注。

C越来越普遍地同多种类型的计算机(有时称为硬件平台)配合使用,最终衍生了C的多种变体。各种变体大体类似,但往往互不兼容。如程序开发者需要编写可移植的程序,以便能在多种平台上运行,这无疑是一处致命的缺陷。一个越来越清楚的事实是,C迫切需要一个标准化的版本。1983年,由美国国家计算机和信息处理标准协会(X3)正式成立了X3J11技术委员会,目的是“提供一个无歧义的、与机器无关的语言定义”。1989年,标准被正式通过。ANSI与国际标准化组织(ISO)协作,致力于C在全世界范围内的标准化普及工作;联合标准文档出版于1990年,所以也称为ANSI/ISO 9899:1990。这份文档的副本可在ANSI订购。Kernighan和Ritchie合著的第2版出版于1988年,详细讲述了ANSI C版本语言,这是目前全球使用最广的语言版本之一。

**可移植性提示 1.1** 由于C是一种标准化的、与硬件无关的、使用较广的语言,所以用C编写的应用程序通常无需修改,或只需稍作修改,即可用于多种不同的计算机系统。

C++是对C的一个扩展,它是Bjarne Stroustrup于20世纪80年代初在贝尔实验室开发成功的。C++提供了数量众多的特性,目的是对C语言进行改进。但更重要的,它还引入了崭新的面向对象编程的能力。

软件工业正在酝酿一场革命。快速、正确和经济地构建软件——这一直都是一个难以实现的目标。每次需要新的、更强人的软件时,人们都会以这个目标为准绳。对象本质上是一种可重复使用的软件组件,用于对现实世界中的各种物件进行建模。软件开发者发现,与以前采用的流行编程技术(如结构化编程)相比,假如采用一种模块化的、面向对象的设计及实现方法,那么可显著提高软件开发小组的效率。面向对象的程序更易理解、纠正和修改。

另外,人们还开发了其他许多面向对象的语言,其中包括施乐的Palo Alto研究中心

(PARC)开发的 Smalltalk。Smalltalk 是一种纯粹的面向对象语言——一切都是对象。C++ 则属于一种混合型语言,既可采用传统的 C 风格进行 C++ 编程,也可采用新的面向对象风格,甚至还能在同一个项目中混合采用两种风格。1.9 节还将讨论令人激动的、以 C 和 C++ 为基础的新一代语言——Java。

## 1.8 C++ 标准库

C++ 程序由一系列类和函数构成。你可自行编写构成一个 C++ 程序所需的每一个组件。但大多数 C++ 程序员都直接利用由 C++ 标准库提供的现成的、丰富的类和函数。因此,要想认识 C++ 世界,实际只需要学习两个方面的知识。一是学习 C++ 语言本身;二是学习如何使用 C++ 标准库中提供的类和函数。对这些类和函数的讨论将贯穿全书。要想深入理解包括在 C++ 中的 ANSI C 库函数,如何实现它们,以及如何用它们编写可移植的代码,就有必要参阅 Plauger 编著的书。标准类库通常由编译器厂商提供。大多数特殊用途的类库由独立的软件开发商提供。

**软件工程知识 1.1** 使用“组装方法”创建程序,尽量不重复别人已经做过的工作,尽可能利用现有的构建单元——这正是“软件重用”的主旨,也是面向对象编程的精髓所在。

**软件工程知识 1.2** 进行 C++ 编程时,通常要使用以下基本构建单元:来自 C++ 标准库的类和函数、你自行创建的类和函数以及由一些知名的第三方库所提供的类和函数。

自行创建函数和类的优点在于:你可确切地知道它们是如何工作的,以便自己能对 C++ 代码进行全面检查。但缺点在于:在设计、开发和维护新函数和类时,为保障其正确性和高效率地工作,需要花费更多的时间与精力。

**性能提示 1.1** 采用标准库的函数和类,而不是自行编写相应的版本,可有效提高程序性能,因为这些组件经过精心的编写,可保证高效而正确地运行。

**可移植性提示 1.2** 采用标准库函数和类,而不是编写自己的相应版本,可显著改善程序的可移植性。这是由于在几乎所有的 C++ 实现方案中,都已包含了这些标准库函数和类的缘故。

## 1.9 Java 和《Java 程序设计》

许多人都认为,微处理器即将发挥重要作用的一个主要领域是智能化的消费类电子设备。考虑到这方面的因素,Sun Microsystems 公司于 1991 年投资启动了一个内部研究项目,代码名为 Green。该项目最终的结果便是诞生了一种基于 C 和 C++ 的语言。该语言的创始人 James Gosling 把它命名为 Oak(橡树)——名字来源于其办公室窗外的一棵橡树。事后不久,在他和几个同事去了本地一家咖啡屋之后,便将其更名为 Java,并沿用至今。

但 Green 项目遭遇了几方面的困难。首先,智能消费类电子设备市场的发展并不如 Sun 想象的那么快。更糟的是,Sun 当时力争的一份大订单被另一家公司抢走了。这个项目可谓举步维艰,随时都可能被取消。但从 1993 年开始有了转机,万维网开始爆炸性地增长,

Sun 几乎立即看到了用 Java 在 Web 页中创建所谓动态内容的巨大潜力。

Sun 在 1995 年 5 月的一次展览会上正式发布了 Java。尽管当时并没有多大反响,但商业界立即对它产生了兴趣,这是因为大家对万维网的兴趣正浓。现在,Java 已经可用来创建包含动态和交互式内容的 Web 页、可开发大型企业级应用程序、可改进 Web 服务器(提供 Web 浏览器中所见的内容的计算机)的功能、可为消费类设备(比如手机、传呼机和个人数字助理等)提供应用程序等等。

1995 年,我们非常仔细地分析了 Sun Microsystems 的 Java 开发过程。1995 年 9 月,我们应邀出席了在波士顿召开的一次因特网会议。Sun Microsystems 的一位代表对 Java 进行了令人振奋的演示。在这个过程中,我们越来越清楚地意识到,Java 必将在交互式多媒体 Web 页的创建中扮演重要角色。与此同时,我们还感受到了该语言的其他更为出色的潜力。

我们认为,对于在校学生来说,Java 非常适用于第一年的编程语言学习,可通过它掌握图形、图像、动画、声音、视频、数据库、连网、多线程和协作式计算的基础。为迎接 1996 年秋季的新学期,我们及时出版了《Java How to Program》(第 1 版)。该书第 3 版于 1999 年出版。

除了用 Java 开发基于因特网和 Intranet 的应用程序之外,还可用它为网络通信设备(比如手机、传呼机和个人数字助理)开发相应软件。不久的将来,你也许会发现,家中的新款立体声音响和其他电器也要通过 Java 技术实现连网。

## 1.10 其他高级语言

迄今为止,人们已开发了数百种高级语言。但是,其中只有几种获得了人们的广泛认可。1954~1957 年,IBM 公司成功开发出 FORTRAN(FORMula TRANslator,公式翻译器),用于简化科学与工程应用中涉及的复杂数学计算。目前,FORTRAN 在全球各地的应用仍然非常广泛——尤其是工程计算领域。

COBOL(COMmon Business Oriented Language,标准商务语言)最早由一系列计算机制造商、政府部门及计算机企业级客户于 1959 年联合开发成功。COBOL 主要用于商业应用。商业往往要求精确而高效地处理大量数据。如今,几乎有一半的商业软件仍采用 COBOL 编写的。

Niklaus Wirth 教授开发的 Pascal 与 C 同时期设计出来的,主要应用于学术界。我们将在下一节详细讨论 Pascal。

## 1.11 结构化编程

20 世纪 60 年代,许多大型软件开发项目都面临严重困境。项目往往不能按时完成,实际成本常常超过预算,最终产品也变得极不稳定。人们逐渐意识到,软件开发的复杂程度远远超过自己的想象。因此,通过 20 世纪 60 年代发起的一系列研究,最终促成了结构化编程概念的问世。这是一种比较严格的程序编写机制,用它可方便地写出思路清晰、高度准确、易于测试/调试以及易于修改的程序。第 2 章将讨论结构化编程的原理。第 3~5 章则会实际开发大量结构化程序。

这一理念的直接成果便是著名的 Pascal 语言的问世,它最早由 Niklaus Wirth 教授于 1971 年开发成功。布雷斯·帕斯卡是 17 世纪法国著名的数学家和哲学家,Pascal 语言便是用他的名字来命名的。该语言的主要设计宗旨是在学术环境中向学生们讲授结构化编程的概念。由于其优秀的设计,很快就成为许多大学流行的入门级编程语言。但令人遗憾的是,由于这种语言缺乏一些关键特性,并不适用于商业、工业和政府等行业。因此,在学校之外的其他地方,几乎很少把 Pascal 作为正式开发语言。

Ada 是在美国国防部(DOD)赞助下开发出来的一种语言——从 20 世纪 70 年代到 80 年代初,开发时间长达数十年。在这之前,美国国防部其实已使用了大量语言,它们均用于构建国防部那套庞大的命令和控制软件系统。然而,DOD 迫切希望专门有一种语言能满足自己的所有需求。最初开始挑选的开发基础是 Pascal,但最后却采用了与其有显著差异的 Ada 语言。Ada 得名于 Ada Lovelace 女士,她是英国著名诗人拜伦的女儿。Ada 女士因为在 19 世纪初编写了全世界第一个计算机程序(用于 Charles Babbage 设计的分析引擎机械计算设备)而声名鹊起。Ada 语言的一项重要功能便是多任务特性;利用多任务特性,程序员可设定让多种事件并行处理。而对我们讨论的其他高级语言(包括 C 和 C++)来说,程序每次都只能处理一个事件。

## 1.12 关键的软件趋势:对象技术

本书作者之一 H. M. Deitel,对于 20 世纪 60 年代那些软件开发公司所遭受的巨大挫折仍然记忆犹新。其中,处境特别糟糕的是那些从事大型项目开发的公司。毕业前夕,H. M. Deitel 有机会在一家行业内顶尖的计算机公司暑期打工,他所在的那个部门负责时间共享、虚拟内存操作系统的开发。对一名大学学生来说,这无疑是一种极其令人兴奋的体验。但在 1967 年夏天,公司宣布不再继续开发一个商业项目。当时,该项目已由数百名技术人员开发了许多年的时间。那时开发软件的难度相当大,软件太复杂。

近年来,硬件成本已显著下降,个人计算机变成了一种“普通家电”。但令人遗憾的是,软件开发的成本不降反升,因为程序员们开发出了越来越强大、越来越复杂的应用程序。与此同时,软件开发的基本技术并没有革命性的进步。本书将介绍学习多种软件开发方法,它们能在一定程度上减少软件开发的成本。

软件工业正在酝酿一场革命。快速、正确和经济地构建软件——这一直是一个难以实现的目标。每次需要新的、更强大的软件时,人们都会以这个目标为准绳。对象本质是一种可重复使用的软件组件,用于对现实世界中的各种物件进行建模。软件开发发现,与以前采用的流行编程技术(如结构化编程)相比,假如采用一种模块化的、面向对象的设计及实现方法,那么可显著提高软件开发小组的效率。面向对象的程序更易理解、纠正以及修改。

20 世纪 70 年代,随着结构化编程(以及结构化系统分析与设计这一相关学科)的问世,人们就已开始对软件技术进行某种程度的改革。但是,直到 80 年代而面向对象编程技术才得以快速推广,到 90 年代才得到人们的广泛认同,软件开发最终才拥有了称心如意的工具,可开始大刀阔斧地改革软件开发过程。

实际上,对象技术最早可追溯到 20 世纪 60 年代中期。C++ 编程语言是由 AT&T 的

Bjarne Stroustrup 在 80 年代初开发成功的。它同时建立在两种语言(C 和 Simula 67),前者最初由 AT&T 针对 UNIX 操作系统的设计在 70 年代初开发成功;后者是 1967 年在欧洲开发成功的一种模拟编程语言。C++ 吸收了 C 的大多数功能,并增加了 Simula 创建和操纵对象的能力。起初,不管 C 还是 C++ 都没有打算广泛用于 AT&T 研究实验室之外的地方,但是,人们很快为这两种语言开发了基本支持。

那么,到底何为对象,它们有何特殊之处? 实际上,对象技术是一种打包方案,帮助我们创建有意义的软件单元。许多软件单元都高度集中在特定的应用领域上。于是,人们见到了品种齐全的日期对象、时间对象、支票对象、发票对象、声音对象、视频对象、文件对象、记录对象等等。事实上,世界上的任何一个名词都用对象来表示。

我们生活在一个对象的世界里。环顾四周便可体会到这一点。汽车、飞机、人、动物、建筑、红绿灯、电梯等等,一切都是对象。在面向对象的语言问世之前,所有的程序语言(如 FORTRAN、Pascal、Basic 和 C)都将重点放在行动(动词)上,而不是放在物件或对象(名词)上。程序员生活在一个对象的世界中,但却不得不使用计算机,主要用动词来进行编程。这种对思维模式的强制转变,加大了编写程序的难度。但现在,由于可选用流行的面向对象语言(比如 Java 和 C++ 以及其他许多语言),程序员可继续生活在一个充斥着形形色色对象的世界,并在使用计算机时,采用自然的、面向对象的方式进行编程。这意味着他们能采用由接触世界而直接感知到的方式编写程序。与程序化编程相比,这样做显然更加自然,而且可显著提高工作效率。

对于程序化编程来说,它的一个主要问题在于,程序员创建的程序单元不能方便而有效地对应现实世界的实体。因此,它们的重用性较差。最普遍的情况是,程序员的每个新项目都得“从头开始”,从头编写非常相似的程序。大量的重复劳动,浪费了大量宝贵的时间和金钱。相反,采用对象技术,只要设计得当,那么软件实体(叫做对象)一经创建,就可极其方便地将其重复用于未来的项目。采用类似 MFC(Microsoft 基础类)、由可重用组件构成的库,以及那些由 Rogue Wave 以及其他许多软件开发商提供的可重用组件,可显著加快特定类型的系统的实施速度(否则还需在新的项目中重新实现这些功能)。

有的公司报告自己在采用了面向对象编程技术之后,软件重用所带来的效益并不大。相反,他们发现采用了面向对象编程技术后,用它写出来的程序更容易理解,更加规范而且更容易维护、修改和调试。出现这一现象基于这样一个众所周知的事实:根据估计,软件成本的 80% 都和原先开发软件时付出的努力无关,而是在软件开发成功之后的生命期内,同后期的升级及维护工作紧密联系在一起。

不管面向对象技术最终带来的是哪方面的效益,摆在面前的事实是,在以后的几十年间,面向对象编程无疑会成为最关键的一种编程方法。

**说明:**我们将在正文中列举许多这样的软件工程知识,以解释会影响和改进一个软件系统(尤其是大型软件系统)总体结构及质量的关键性概念。另外,我们还会强调一些良好编程习惯(帮助你采用更清晰、更容易理解、更易维护以及更易测试和调试的方式进行编程)、常见编程错误(指出一些应注意的问题,帮助你避免在自己的程序中出现这些问题)、性能提示(一些特殊的技巧,使你的程序运行得更快,并占用较少的内存)、可移植性提示(帮助你写出能在多种计算机平台上运行的程序,同时无需修改或仅需少量修改)以及测试和调试提

示(帮助你找出程序中的错误,更重要的是教你如何在第一时间避免错误)。许多这样的技术和准则都属于一些指导性原则;毫无疑问,你可根据需要,开发自己偏爱的编程风格。

自行创建代码的优点在于可准确知道其工作原理,可对代码进行检查。缺点在于设计和开发新代码往往非常花时间,而且较为复杂。

**性能提示 1.2** 尽量重复使用现有代码组件,而不是去开发自己的版本,这将有助于提高程序性能,因为这些组件通常在当初编写时就已考虑到了执行效率的问题。

**软件工程知识 1.3** 一系列内容全面的可重用软件组件类库可通过因特网和万维网获得。类似的许多库都是免费的。

## 1.13 典型C++环境的基础

C++ 程序通常由 3 部分构成:程序开发环境、语言本身以及C++ 标准库。在后面的讨论中,我们对图 1.1 展示的一个典型C++ 程序开发环境进行了解释。

C++ 程序通常要经历 6 个执行阶段(见图 1.1),其中包括:编辑、预处理、编译、连接、装入和执行。我们在此以一个典型的基于 UNIX 的C++ 系统为例(注意:本书的程序只需做很少的修改,或者根本无需修改,便可在大多数最新的C++ 系统上运行,包括基于Microsoft Windows 的系统在内)。如果你使用的不是 UNIX 系统,请参阅自己的系统手册,或者询问你的导师,了解如何在自己的开发环境中完成这些任务。

第一个阶段的任务是编辑一个文件。这是用一个编辑器程序来完成的,程序员用编辑器输入一个C++ 程序。如有必要,还要纠正其中的错误。随后,程序保存到一个辅助存储设备中,比如存到磁盘上。C++ 程序文件通常带有 .cpp、.cxx 或 .C 扩展名(注意 C 是大写的)。

请参见C++ 环境的用户文档,深入了解文件扩展名。UNIX 系统中,目前最常用的两个编辑器是 vi 和 emacs。个人计算机中,像 Borland C++ 和 Microsoft Visual C++ 这样的C++ 软件包都配备内置的编辑器,它们同编程环境紧密集成。在此,须具备一定的编程知识。

接着,程序员发出编译程序命令。编译器负责将C++ 程序翻译为机器语言代码(或目标代码)。C++ 系统中,在编译器开始翻译之前,会自动执行一个预处理程序。C++ 预处理程序(又称预处理器)将遵循一些名为预处理程序指令的特殊命令,它们指示在编译开始之前,应先采取一些特定的行动。这些行动通常包括:将其他文本文件包括到要编译的文件中,以及执行特定的文字替换等等。在本书前几章,会讲解一些最常用的预编译指令;对于所有预编译指令的详细讨论参见“预处理”一章。预处理器会在程序翻译成机器语言之前,由编译器调用。

下一阶段叫做连接。C++ 程序通常包含了对其他地方定义的函数的引用,比如在标准库或私有库(由一组程序员在处理一个特定的项目时创建)中定义的函数。如缺失了这些组件,由C++ 编译器生成的目标码通常会包含“漏洞”。连接器的功用便是将目标码同缺失函数的代码连接起来,以生成一个可执行映像(其中不再有缺失的部分)。在一个典型的基于 UNIX 的系统中,用于编译和连接一个C++ 程序的命令是 CC。为编译和连接一个名为 welcome.C 的程序,需键入命令

```
CC welcome.C
```

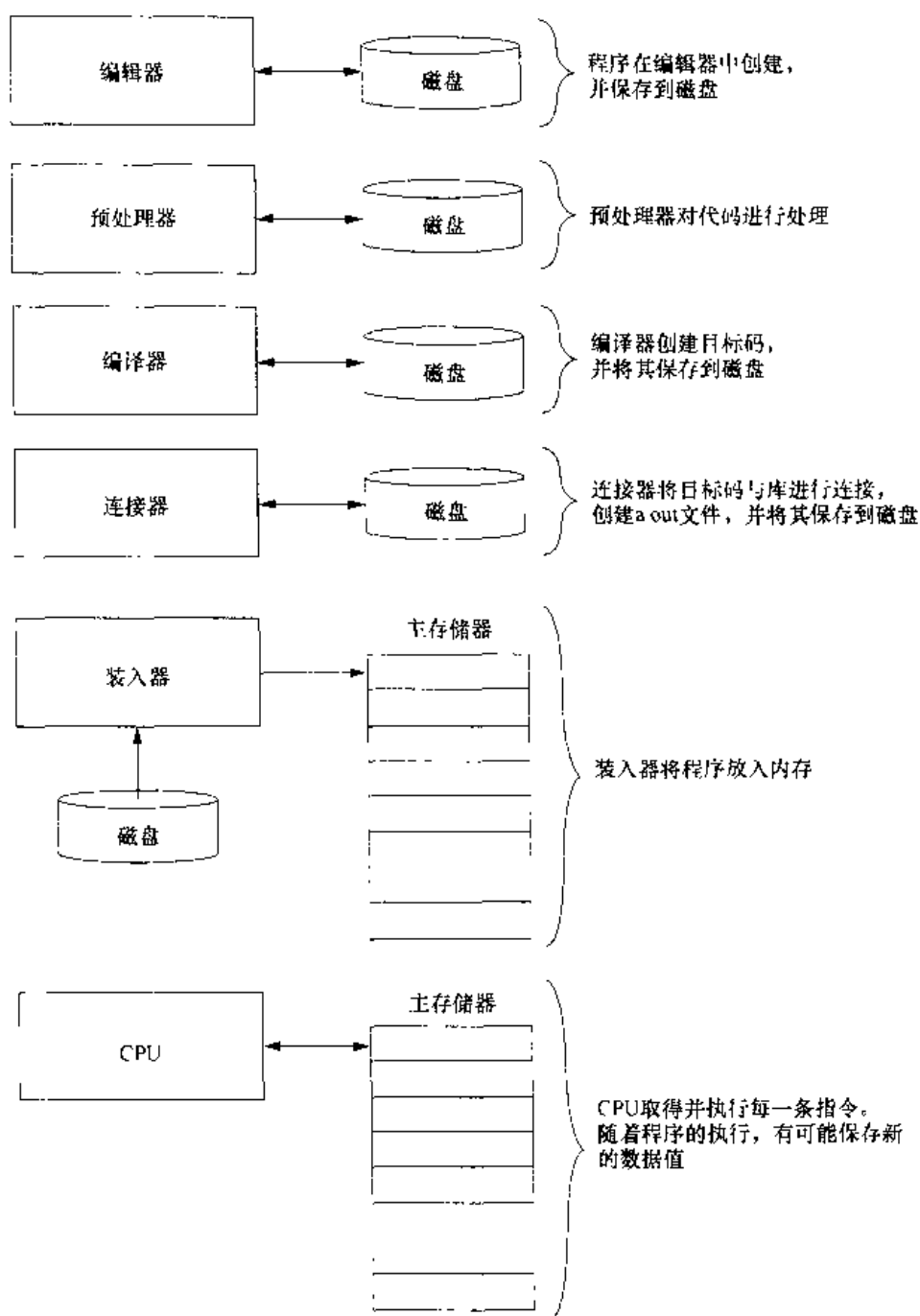


图 1.1 一个典型的C++ 环境

上述命令在 UNIX 提示行中输入,完成后按回车键即可。如程序正确地编译和连接,便会生成一个名为 a.out 的文件。它是 welcome.C 程序的可执行映像。

下一个阶段是装入。程序为了能够执行,首先必须进入内存。这是通过装入器来完成的。它的作用是从磁盘上取得可执行映像,然后将其传输进入内存。如果还需要来自共享库的其他组件为程序提供支持,那些组件也会同时装入。

最后,计算机在其 CPU 的控制下,以每次一条指令的形式,开始执行程序。为了在 UNIX 系统上装入执行程序,我们在 UNIX 提示行中键入 a.out,再按回车键。

初次试验时,程序不一定能成功执行。由于各种我们以后会讨论的错误,上述每一个阶



段都可能出错。例如,一个正在执行的程序可能试图被零除(如同数学运算一样,在计算机中也属于非法操作),这会导致计算机打印一条出错消息。程序员随后必须到编辑阶段进行必要的纠正,然后重复后续阶段,判断自己的纠正能否见效。

**常见编程错误 1.1 “被零除”**这样的错误是在程序运行时发生的,所以这类错误叫做运行期错误或者执行期错误。被零除通常被认为是一种致命错误;也就是说,是一种会导致程序立即中止,无法成功完成其工作的错误。非致命错误则允许程序运行至结束,但通常会产生不正确的结果(注意:在某些系统上,被零除并不是致命错误。详情可参见系统文档)。

大多数C++ 程序都要进行数据的输入和/或输出。有的C++ 函数会从 `cin`(标准输入流;发音与“see-in”同)取得自己的输入值。最常见的 `cin` 便是键盘,但它也可能同另一个设备连接。数据则通常输出至 `cout`(标准输出流;发音与“see-out”同),这通常是计算机屏幕,但 `cout` 也可同另一个设备连接。当我们说程序“打印”一个结果时,通常是指在屏幕上显示结果数据,但也可以是指输出至其他设备,比如磁盘和硬拷贝打印机等等。另外,还有一个所谓的标准错误流,我们称其为 `cerr`。`cerr` 流(通常同屏幕连接)用于显示出错时的信息提示。用户们经常会对常规输出数据进行重定向,例如将 `cout` 定向屏幕外的另一个设备;与此同时,继续将 `cerr` 分配给屏幕,以便能在出现错误时立即看到提示。

## 1.14 硬件发展趋势

在硬件、软件和通信技术方面,程序开发社区会不断地对它们施加重要影响,促进它们的发展。每年,大多数产品及服务的价格都会有所上升。相反,计算机和通信领域的成本却在逐渐下降——特别是为这些技术提供支持的硬件的成本。数10年来,而且在可以预见的将来,硬件成本一直呈急剧下降态势。这便是技术的奇迹,也是促进当前经济繁荣的另一个重要因素。每过一两年,计算机的容量(特别是内存容量——程序需要放在内存中执行)、辅助存储的容量(比如磁盘存储容量,用于长时间保存程序和数据)以及处理器的速度(决定了程序的执行速度)都会大致翻一番。通信领域的发展同样如此迅猛,同时成本也在不断降低。特别是这些年来,随着人们对通信带宽的需求越来越大,通信市场的竞争愈来愈激烈,也直接造成了通信价格的大幅下降。据我们所知,在其他任何领域,都不存在技术越来越先进,成本却下降得如此之快的情况。

20世纪60年代和70年代,随着计算机的广泛使用,许多人开始憧憬计算和通信技术会大大提高人们的生产效率。不过,就当时的情况来看,这样的梦想并没真正实现。许多单位花大价钱购置了庞大的计算机,而且肯定用最优的方式来使用它们,但生产效率并不能得以显著提高。这种情况一直延续到70年代末和80年代,直到人们发明了微处理器芯片技术。应用了这种技术后,90年代的生产效率大幅提高才有了一个坚实的基础,它也是促成目前经济繁荣的重要因素。

## 1.15 因特网发展简史

20世纪60年代末期,本书作者之一 H. M. Deitel 已从麻省理工学院(MIT)毕业。当时,

他从事研究工作的单位是 Project Mac, 目前已成为 MIT 计算机科学实验室, 也是万维网协会 (W3C) 的总部——W3C 是由 ARPA (美国国防部高级研究项目局, Advanced Research Projects Agency of the Department of Defense) 组建的。当年, ARPA 召开了一个学术研讨会。几十位曾获得 ARPA 奖学金的毕业生聚集于伊利诺州大学, 交流学术经验。在这次会议上, ARPA 宣布了一项惊人计划, 打算将自己出资赞助的大学及研究所的主计算机系统连接成一个网络。而且最关键的是, 计算机之间的连接速度将达到当时看来足以令多数人震惊的 56KB (亦即每秒 56 000 位, 或如今 56K 调制解调器的速度)。为什么会如此多人震惊呢? 因为当时普通人通过电话线的连接速度最高只能达到每秒 110 位! 时隔多年, H. M. Deitel 先生对当年会议上的热烈场面依然记忆犹新。哈佛大学的研究人员开始谈论以后如何“跨越全国”, 同犹它州州立大学的“超级计算机”Univac 1108 进行通信, 以便加快本校计算机图形研究的速度。当然, 其他大量可能的、令人心驰神往的应用也崭露头角。在人们的印象中, 学术研究似乎能立即向前跨出一大步。这次会议后不久, ARPA 果然没有食言, 很快开始建设网络——人们很快便知道这个网络称为 ARPAnet (阿帕网)——它就是因特网的始祖!

不过, 网络建成后, 同前期计划稍有偏差的是, 尽管 ARPAnet 确实允许研究人员相互间共享计算机, 但事实证明, 它最主要的用途却是通过电子邮件进行快速和简便的通信。在如今的因特网上, 电子邮件仍然是非常重要的一项应用。电子邮件使全球数亿人之间的通信变得异常简便和高效。

按照 ARPA 的设计要求, 这个网络的一项主要目标是让多个用户通过相同的通信线路 (如电话线) 同时收发信息。网络采用一种名为包交换的技术运行, 其中, 数字化的数据是用一系列小包 (名为数据包, 或简称包) 来传送的。在数据包内, 包含有数据、地址信息、错误控制信息以及排队 (顺序) 信息。利用地址信息, 数据包可正确路由到它们的目的地。排队信息则用于重新装配数据包 (由于网络及路由的复杂和不确定因素, 数据包的抵达顺序可能和发出时有所区别), 使它们按原来的次序传送, 以便接收者能正确接收。在同一条线路上, 完全可能混合传送多个数据包。和其他那些专用于通信的线路相比, 包交换技术可显著减少数据传输成本。

ARPAnet 在设计时, 已考虑到今后不会再采用中心控制的形式。也就是说, 即使网络的某部分瘫痪了, 其余部分仍能通过其他路径, 将数据包从发送端路由传到接收端。

在 ARPAnet 上, 用于通信的协议叫做 TCP (传输控制协议)。TCP 可保证数据从发送端正确路由到接收端。这样一来, 收到的数据包肯定是原封未动的。

世界各地, 与该因特网雏形并行发展的还有许多单位自行开发的网络。这些网络在单位内部使用, 或在不同的单位之间使用。当时, 许多网络硬件和软件都开始有所进展。不过, 由此也带来了一个十分紧迫的问题, 谁来制订统一的标准, 把所有这些东西都结合到一起, 实现彼此间的自由通信呢? 明智的 ARPA 迅速研制了著名的 IP 协议——即“网际协议” (Internet working Protocol), 从而创造出了真正的“网间网”的概念——也为目前的因特网打下一个坚实的基础。如将 TCP 和 IP 合到一起, 便是大家耳熟能详的 TCP/IP 协议。

最初, 因特网只限于在大学和学术研究机构中使用。不久, 军方成为它最大的用户。最后, 政府决定向商业领域开放因特网。有趣的是, 当年许多研究机构及军事单位的开发人员

还为政府的这项决定而愤愤不平——他们担心随着用户数量的增加,自己的研究工作将大受干扰。不管怎样,因特网再也不像以前那么“单纯”了。

事实证明了一切。与许多研究人员事先想象的相反,商业用户很快便意识到,有效利用因特网,应该能改进公司的运作,并为客户提供更新、更好的服务。为此,他们投入了大量资金,不断开发与改进因特网。与此同时,不管在商业电缆公司之间,还是在硬件及软件开发与生产公司之间,都展开了激烈的竞争。大家殚精竭虑,为因特网的明天出谋划策——其目的是试图占领未来的市场。尽管如此,用户总是受益的一方。最后的结果便是,因特网的带宽(指通信线路的信息负载能力)得到了极大的提高,使用费用却在急剧下降。不管美国还是其他一些国家,经济上的繁荣在很大程度上都得益于因特网的迅猛发展。过去 10 年,他们已在这个领域中尝到了极大的甜头;而且在可以预见的将来,还将继续受益!

## 1.16 万维网发展简史

万维网(World Wide Web)使计算机用户能在因特网上查找并游览基于多媒体的、几乎涉及所有主题的文档。更妙的是,在这些文档中,可同时包含文字、图形、动画、声音以及视频信息。因特网问世于 30 年前,但万维网其实是在 20 世纪 90 年代问世的。1990 年,CERN(欧洲粒子物理实验室)的 Tim Berners-Lee 开发了万维网的原型,同时还开发了几个用于提供底层支持的协议。

因特网和万维网无疑可算作人类最重要的、意义最深刻的少数几项发明之一。过去,大多数计算机都只能采用“单机”模式运行——相互间不能连接。但对今天的许多应用来说,却可轻易地同全球数亿台计算机通信。在因特网中,混合运用了计算和通信等领域的多项技术。它使我们的工作更轻松;使世界各地的人们更容易、更快速地取得信息;使个人及小公司向全世界展示自己;它改变了许多商业规则;人们可搜索几乎任何产品或服务的最佳价格,而且可以马上获得需要的产品和服务;有特殊兴趣的人马上就可以找到志同道合者;研究人员可随时获知全球最新的学术进展。

## 1.17 C++ 和本书的常规注意事项

C++ 是一种复杂的语言。有经验的 C++ 程序员有时会因为自己能够以某种古怪、繁复而又令人困扰的方式使用这种语言而沾沾自喜。但这并不是一种良好的编程习惯,因为这会使程序更难阅读、更容易表现失常、更难测试和调试以及难以满足不断变化的需求。本书是为刚入门的程序员而编写的,所以会着重强调程序的清晰性。

**良好编程习惯 1.1** C++ 程序应以简单和直接的方式编写。这有时也称为 KIS(“keep it simple”的简称,即“尽量简单”)编程方法。千万不要去尝试一些古怪的用法,滥用这种语言。

恐怕大家已经知道,C 和 C++ 均是可移植的语言,用 C 和 C++ 写的程序可在多种不同的计算机上运行。但是,完美的可移植性是一个难以实现的目标。ANSI C 标准文档列出了一系列非常多的可移植性问题,另外还有许多书只讨论可移植性。

**可移植性提示 1.3** 尽管有可能写出可移植的程序,但在不同的 C 和C++ 编译器以及不同的计算机之间,会产生许多问题。这些问题的存在,加大了移植程序的难度。并不是说用 C 和C++ 来写程序就能保证移植性。程序员通常需要直接同特定的编译器和计算机平台打交道。

写作本书时,我们通读了 ANSI/ISO C++ 标准文档,并审核了内容,以检验其完整性和准确性。然而,C++ 是一种具有丰富特性的语言,语言中的一些细节,以及一些较高深的主题,是我们未曾涉及到的。如果需要有关C++ 的更多技术细节,建议你阅读C++ 标准文档 C++ 标准文档可从 ANSI 的 Web 站点处订购:<http://www.ansi.org/>。

文档的标题是“Information Technology – Programming Language – C++”,文档编号为“ISO/IEC 14882 – 1998”。如果不愿意购买,也可直接在下述 Web 站点处查阅标准的草案版本:<http://www.cygnus.com/misc/wp/>。

我们提供了一个C++ 资源附录,其中列举了大量因特网和万维网站点,它们都同C++ 和面向对象编程有关。

当前版本C++ 的许多特性与早期的C++ 实现不兼容,所以本书的一些程序可能不能在早期的某个C++ 编译器上工作。

**良好编程习惯 1.2** 仔细阅读你所用C++ 版本的用户手册。经常查阅这些手册,才能确保自己能了解并正确使用C++ 丰富的特性。

**良好编程习惯 1.3** 计算机和编译器是最好的老师。仔细阅读了C++ 语言手册之后,如果不能确定一项C++ 的特性是如何工作的,不妨用一个短小的“测试程序”进行试验,观其效果。设置编译器选项,令其报告“最多的警告”。研究程序编译时出现的每一条错误消息,并对程序进行纠正,最终去除这些消息。

## 1.18 C++ 编程简述

C++ 语言提供了一个结构化、有序的方式来进行计算机程序设计。我们现在打算介绍C++ 编程,并展示几个例子,对C++ 的一些重要特性进行阐述。针对每个例子,都会采取每次分析一条语句的办法。在第2章,我们会详细介绍C++ 中的结构化编程。然后,我们会将结构化编程方式一直沿用到第5章。自第6章起,我们将学习C++ 的面向对象编程概念。同样地,由于面向对象编程在本书中的重要性,所以前5章都会以“对象思想”小节结尾。这些特殊的小节介绍了重要的面向对象概念,并展示了一个案例分析,帮助读者设计和实现一个实际的面向对象C++ 程序。

## 1.19 一个简单的程序:打印一行文字

C++ 采用的记号对非程序员来说可能相当陌生。首先,让我们来思考一个简单的程序,它的目的是打印一行文字。该程序及其屏幕输出如图 1.2 所示。

```
1 //Fig. 1.2: fig01_02.cpp
```

```

2 //A first program in C++
3 #include <iostream>
4
5 int main()
6 {
7     std::cout << "Welcome to C++! \n";
8
9     return 0; //indicate that program ended successfully
10 }

```

输出结果:

Welcome to C++!

图 1.2 文字打印程序

该程序展示了C++语言的一些重要特性。下面将逐行探讨该程序。第1行和第2行

```

//Fig. 1.2; fig01_02.cpp
//A first program in C++

```

两行均以//开头,这表明//后的内容均为注释。程序员在编写程序时加注注释,以便为程序制作文档,并增强程序的可读性。注释还有助于其他人阅读和理解你所编的程序。程序运行时,注释不会导致计算机对其采取任何操作。C++编译器会忽略注释,根本不会为其生成任何机器语言目标码。其中,“A first program in C++”这一注释只是为了说明该程序的用途。以//开头的注释叫做单行注释,因为注释将在当前行的末尾中止(注意:C++程序员还可使用C的注释风格;也就是说,注释中可以包含多行文字——以/\*开头,以\*/结尾)。

**良好编程习惯 1.4** 每个程序都应以注释开头,以描述该程序的用途。

第3行

```
#include <iostream>
```

是一条预处理程序指令;换言之,是发送给C++预处理器的一条消息。以#开头的行会在程序编译之前由预处理器进行处理。这个特殊的代码行用于指示预处理器,在程序中包含<iostream>这一输入/输出流头文件的内容。对于任何程序,只要它需要采用C++风格的流输入/输出,将数据输出至屏幕,或从键盘输入数据,便应将上述头文件包括在内。大家稍后将看到,图1.2的程序要将数据输出至屏幕。iostream的内容将在以后详细解释。

**常见编程错误 1.2** 在一个需要从键盘输入数据,或者需要将数据输出至屏幕的程序中,假如忘了包括iostream文件,会导致编译器报告一条错误消息。

第5行

```
int main()
```

是每个C++程序都有的一部分。main之后的圆括号指出main是一个基本的程序构建单元,即所谓的函数。C++程序可包含一个或多个函数,但其中一个必须是main。图1.2只包含了一个函数;注意C++程序无论如何都会从函数main处开始执行,即便main并不是程序中列出的第一个函数。main左边的int关键字指出main会“返回”一个整数值。等到第3章,当我们深入学习函数时,还会解释对一个函数而言,“返回一个值”的真实含义。目前只需记住每个程序的main函数的左侧都要加关键字int。

第6行的左花括号( { )必须用于每个函数主体的开始处。一个对应的右花括号( } )则必须用于每个函数主体的结束处。

#### 第7行

```
std::cout << "Welcome to C++! \n";
```

用于指示计算机在屏幕上打印包含在引号之间的字符串。注意对于整行代码而言——其中包括 `std::cout`、`<<` 操作符、“Welcome to C++! \n”字符串以及最后的分号( ; )——则称为语句。每条语句都必须以一个分号结尾(分号也叫做语句中止符)。在C++中,输出和输入是通过字符流来完成的。因此,当上述语句执行后,它会将 Welcome to C++! 这一字符流发送给标准输出流对象,即 `std::cout`。后者往往同屏幕“连接”。我们将在第11章详细讲解 `std::cout` 的许多特性。

注意我们将 `std::` 放在 `cout` 之前。`std::cout` 指出我们打算使用一个名称,在这种情况下是 `cout`,它从属于 `std` 这个名称空间。名称空间是一种高级的C++特性。我们将在第21章深入讨论名称空间。就目前来说,你只需记住每次在程序中用到 `cout`、`cin` 和 `cerr` 时,都在它的前面加上 `std::`。这当然有点儿麻烦——在图1.14中,我们引入了 `using` 语句,它使我们避免在 `std` 名称空间的每一个名称之前,都放置一个 `std::`。

`<<` 操作符是一个流插入操作符。这个程序执行时,操作符右边的值(即右操作数)会插入输出流中。右操作数的字符通常会按它们在双引号之间出现的样子,原封不动地打印出来。但要注意, `\n` 字符是不会在屏幕上打印的。这里的反斜杠( \ )叫做换码操作符。它表明要输出的是一个“特殊”字符。如在一个字符串中遇到一个反斜杠,它的下一个字符便同反斜杠合并到一起,共同构成所谓的换码序列。`\n` 这个换码序列代表一个 `newline`,即“换行符”。它会导致光标(亦即屏幕当前位置说明符)移至屏幕上一行的起始处。图1.3对其他一些常用的换码序列进行了总结。

换码序列	说明
<code>\n</code>	换行符。将屏幕光标定位至下一行起始位置
<code>\t</code>	水平制表位。将屏幕光标移至下一个制表位
<code>\r</code>	回车符。将屏幕光标定位至当前行起始位置;不转到下一行
<code>\a</code>	警告、发出系统响铃声
<code>\</code>	反斜杠。用于打印一个反斜杠字符
<code>"</code>	双引号。用于打印一个双引号字符

图 1.3 一些常用的换码序列

**常见编程错误 1.3** 遗漏语句末尾的分号属于语法错误。一旦编译器不能正确识别语句,便会导致语法错误。编译器通常会发出一条错误消息,帮助程序员定位并纠正不正确的语句。语法错误是对语言规则的违背。语法错误也叫做编译错误、编译期错误或编译时错误(因为它们是在编译时显现出来的)。

#### 第9行

```
return 0; // indicate that program ended successfully
```

包含于每个 `main` 函数的结束位置。C++ 关键字 `return` 是我们退出一个函数的手段之一。如上所示,如在 `main` 的结束位置使用 `return` 语句,0 值就意味着程序已成功中止。第3章还会

详细讨论函数。届时,你会更清楚包含这条语句的原因。此时只需记住每个程序都要包含该语句即可;否则,某些系统上,编译器会产生警告消息。

第 10 行的右花括号(})指出 main 结束。

**良好编程习惯 1.5** 许多程序员让函数打印的最后一个字符是换行符(\n)。这样可保证函数将屏幕光标定位在一个新行的起始处。这样一个自发的约定可促进软件的重用能力——此为软件开发环境的一个重要目标。

**良好编程习惯 1.6** 针对每个函数的主体,令其在定义函数主体的花括号内部,缩进一级位置。这样可使程序的函数结构更清晰,增强其可读性。

**良好编程习惯 1.7** 先为你喜欢的缩进距离拟出一个约定,然后始终坚持这一约定。可考虑用制表位(按 Tab 键)生成缩进。但是,不同的系统上,制表位的距离往往不同。因此,建议你要么使用 1/4 英寸制表位,要么(这是一种更好的做法)用 3 个空格构成一个缩进级别。

Welcome to C++! 可采用几种方式打印。例如,图 1.4 使用了多个流插入语句(第 7 行和第 8 行),但是生成的输出与图 1.2 完全相同。这样做之所以有效,是由于每个流插入语句都会自上一条语句停止打印的地方继续打印。第一个流插入语句打印 Welcome 和一个空格;第二个流插入语句则接在空格之后,在同一行上继续打印。通常,C++ 允许程序员采用多种形式表达语句。

```
1 //Fig.1.4: fig01_04.cpp
2 //Printing a line with multiple statements
3 #include <iostream>
4
5 int main()
6 {
7     std::cout << "Welcome ";
8     std::cout << "to C++! \n";
9
10    return 0; //indicate that program ended successfully
11 }
```

输出结果:

Welcome to C++!

图 1.4 使用 cout,在同一行上打印多条流插入语句

如图 1.5 所示,单独一条语句可使用换行符来打印成多行。每次在输出流中遇到\n(换行符)换码序列时,屏幕光标都会自动定位到下一行的起始位置。要想在输出结果中生成一个空行,只需就像图 1.5 那样,连续使用两个换行符即可。

```
1 //Fig.1.5: fig01_05.cpp
2 //Printing multiple lines with a single statement
3 #include <iostream>
4
5 int main()
6 {
```

```

7   std::cout << "Welcome \nto\n\nC++! \n";
8
9   return 0; // indicate that program ended successfully
10 ;

```

输出结果:

```

Welcome
to
C++!

```

图 1.5 使用 cout, 一条语句可打印为多行

## 1.20 另一个简单的程序: 两个整数相加

下一个程序将使用输入流对象 `std::cin` 以及流提取操作符 `>>`, 获取用户通过键盘输入的两个整数, 计算这两个值的和, 再用 `std::cout` 输出结果。程序及输出结果如图 1.6 所示。

```

1  //Fig.1.6: fig01_06.cpp
2  //Addition program
3  #include <iostream>
4
5  int main()
6  {
7      int integer1, integer2, sum;           //declaration
8
9      std::cout << "Enter first integer\n";   //prompt
10     std::cin >> integer1;                   //read an integer
11     std::cout << "Enter second integer\n";  //prompt
12     std::cin >> integer2;                   //read an integer
13     sum = integer1 + integer2;              //assignment of sum
14     std::cout << "Sum is " << sum << std::endl; //print sum
15
16     return 0;    //indicate that program ended successfully
17 ;

```

输出结果:

```

Enter first integer
45
Enter second integer
12
Sum is 117

```

图 1.6 一个求和程序

注意第 1 行和第 2 行的注释

```

//Fig.1.6: fig01_06.cpp
//Addition program

```

它们指出文件名以及程序的用途。第 3 行的 C++ 预编译指令

```
#include <iostream>
```

用于将 `iostream` 头文件的内容包含到程序中。



如前所述,每个程序都要先执行函数 main。左花括号标记出 main 主体的起始位置,相应的右花括号标记出 main 的结束位置。第 7 行

```
int integer1, integer2, sum;    //declaration
```

它是一个声明。其中, integer1、integer2 和 sum 代表变量的名称。变量是计算机内存中的一个特定的位置,可在此存放一个值,以便由程序使用。该声明规定 integer1、integer2 和 sum 这 3 个变量的类型为 int;换言之,这些变量将用来存储整数值,亦即像 7, -11, 0, 31 914 这样的数字。所有变量在声明时都必须规定一个名称和一个数据类型,然后才能用于程序中。相同类型的几个变量既可一次性地全部声明,也可分别多次声明。我们可以连续写 3 个不同的声明,每次声明一个变量。但显然,上述声明更加简练。

**良好编程习惯 1.8** 有的程序员喜欢一行声明一个变量。采用这种格式,可方便地在每个声明后插入说明性的注释内容。

我们不久还会讨论 double 数据类型(用于指定实数,亦即含有小数点的数字,例如 3.4, 0.0 和 -11.19 等等)以及 char 数据类型(用于指定字符数据;在一个 char 变量中,只能容纳一个小写字母、一个大写字母、一个数位或者一个特殊字符,比如 x, \$, 7, \*, 等等)

**良好编程习惯 1.9** 每个逗号(,)后都应插入一个空格,以增强程序可读性。

一个变量名可为任意有效的标识符。一个标识符可由一系列字符构成,其中包括字母、数位和下划线(\_),但却不能以一个数位开头。C++ 要求严格地区分大小写——大写和小写字母被认为是不同的字符。因此,al 和 A1 完全是两个不同的标识符。

**可移植性提示 1.4** C++ 语言本身允许任意长度的标识符,但你的系统和/或具体的 C++ 实现有可能对标识符的长度进行了某种限制。因此,请使用 31 个字符以内的标识符,以保证可移植性。

**良好编程习惯 1.10** 挑选一个有意义的变量名,将有助于保障程序的“自编档能力”;也就是说,只需读一读程序,即可轻松理解,而不必是必须求助于手册,或使用过多的注释。

**良好编程习惯 1.11** 避免使用以下划线和双下划线开头的标识符,因为 C++ 编译器可能采用这种形式的名称为其内部的某些用途提供服务。这样,有助于避免你选择的名称同编译器选择的名称混淆。

变量声明可置于函数中的任何地方。然而,变量声明必须出现于变量在程序中的实际使用位置之前。例如,在图 1.6 中的程序,可以用多条语句来声明 3 个独立的变量。或者,分别使用单独的声明语句。声明

```
int integer1;
```

可放在下一行

```
std::cin >> integer1;
```

之前。声明

```
int integer2;
```

可放在下一行

```
std::cin >> integer2;
```

之前。声明

```
int sum;
```

则可放在下一行

```
sum = integer1 + integer2;
```

之前。

**良好编程习惯 1.12** 可执行语句之间的声明之前,需插入一个空行。这样可在程序中突出声明语句,使程序更加清晰。

**良好编程习惯 1.13** 如果你喜欢在函数的起始处放置声明,请用一个空行,以便区分声明同函数中的可执行语句,突出声明结束的位置和可执行语句的开始位置。

#### 第9行

```
std::cout << "Enter first integer\n"; //prompt
```

用于在屏幕上打印字符串 Enter first integer(这也称作一个字面字符串,或者一个字面值),并将光标定位到下一行起始的位置。此消息叫做一条提示,因为它的作用是要求用户采取一项特定的行动。我们常将上述语句读作:“cout 获得字符串Enter first integer\n”。

#### 第10行

```
std::cin >> integer1; //read an integer
```

用输入流对象 cin(属于 std 这个名称空间)以及流提取操作符 >>,以便从键盘取得值。流提取操作符同 std::cin 配合使用,目的是取得自标准输入流(通常是键盘)而输入的字符。我们喜欢将上述语句读作:“std::cin 把一个值提供给 integer1”,或者简单地读作:“std::cin 提供给 integer1”。

计算机执行上述语句时,会等候用户为 integer1 变量输入一个值。用户需要输入一个整数(以字符形式),然后按回车键做出响应,这样可将数字发送给计算机。随后,计算机将数字的字符形式转变成一个整数,并将此数字(或值)赋予变量 integer1。在程序中,以后对 integer1 的任何引用都会使用相同的这个值。

std::cout 和 std::cin 流对象简化了用户同计算机之间的交互。由于这样的交互与对话颇为相似,所以通常把它叫做对话式计算或交互式计算。

#### 第11行

```
std::cout << "Enter second integer\n"; //prompt
```

用于在屏幕上打印 Enter second integer 字样,然后定位到下一行的起始处。该语句同样提示用户采取一项行动。第12行

```
std::cin >> integer2; //read an integer
```

用于从用户那里取得 integer2 变量的值。

#### 第13行

```
sum = integer1 + integer2; //assignment of sum
```

是一个赋值语句,用于计算 integer1 和 integer2 这两个变量的和,并使用赋值操作符 =,将结果赋给变量 sum。上述语句一般读作:“sum 取得 integer1 + integer2 的值。”注意,大多数计算都是用赋值语句来执行的。= 操作符和 + 操作符叫做二元操作符,因为它们各自要取用两个操作数。在 + 操作符的情况下,它的两个操作数是 integer1 和 integer2。在前述的 = 操作符的情况下,两个操作数分别是 sum,以及 integer1 + integer2 这个表达式的值。

**良好编程习惯 1.14** 在二元操作符的两端,分别添加一个空格。这样可突出显示操作符,增强程序可读性。

第 14 行

```
std::cout << "Sum is " << sum << std::endl; //print sum
```

用于显示字符串“Sum is”,以及变量 sum 的数值和 std::endl(endl 是“end line”的简称;另外,endl 是 std 这个名称空间中的一个名称)——即所谓的流操纵元。std::endl 这个流操纵元先输出一个换行符,再“刷新输出缓冲”。它的意思可简单地理解为:在某些系统上,输出内容会在机器中临时堆积起来,直到“值得在屏幕上显示”为止。std::endl 可强迫当时任何堆积起来的输出显示到屏幕上。

注意上述语句输出的是多个不同类型的值。流插入操作符“知道”如何输出每一种数据。如果在单独的语句中采用多个流插入操作符(<<),则称做流插入操作的连接(concatenating)、链接(chaining)或连续使用(cascading)。因此,不必用多条输出语句来输出多种数据。

也可在输出语句中直接执行计算。第 13 行和第 14 行的语句可合并为语句

```
std::cout << "Sum is " << integer1 + integer2 << std::endl;
```

这样可避免使用变量 sum。

右花括号(|)用于通知计算机已抵达了 main 函数的结束位置。

C++ 的一个强大的特性在于,用户可创建自己的数据类型(详情将在第 6 章探讨)。这样一来,用户实际可以“教”C++ 如何使用>>和<<操作符,从而输入和输出这些新的数据类型的值(这叫做操作符重载,第 8 章将深入探讨该主题)。

## 1.21 内存的概念

像 integer1、integer2 和 sum 这样的变量名实际对应于计算机内存中的位置。每个变量都有自己的名称、类型、长度和值。

以图 1.6 展示的求和程序为例,执行语句

```
std::cin >> integer1;
```

时,用户键入的字符会转换成一个整数,并将其置入由 C++ 编译器为 integer1 这个名称所分配的一个内存位置。假定用户输入 45 作为 integer1 的值,计算机会将 45 放到 integer1 位置处,如图 1.7 所示。



图 1.7 显示了变量名和值的内存位置

值被置入某个内存位置后,该值会替换该位置原有的值。原有的值会丢失。

仍以求和程序为例,执行语句

```
std::cin >> integer2;
```

时,假定用户输入值 72。该值置于 integer2 的内存位置,如图 1.8 所示。注意这些位置在内存中不一定相邻。

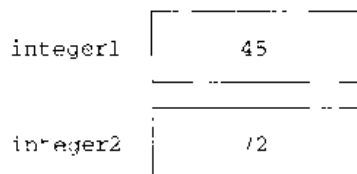


图 1.8 两个变量的值输入之后的内存位置

一旦程序取得 `integer1` 和 `integer2` 的值,会将这些值相加,并将和置于变量 `sum` 中

语句

```
sum = integer1 + integer2;
```

所执行的加法也会代替原先保存在 `sum` 中的值。一旦将 `integer1` 和 `integer2` 的和置入 `sum` 的内存位置,便会发生这样的替换(不管 `sum` 中现有的值是什么,它都会消失)。`sum` 计算好后,内存的情况如图 1.9 所示。注意 `integer1` 和 `integer2` 的值与其计算 `sum` 之前相同。计算机执行完计算后,这些值虽已用过,但未被破坏。因此,我们认为当一个值从某个内存位置读出时,整个过程是非破坏性的。

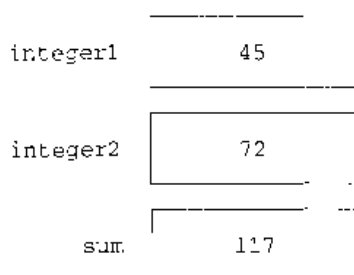


图 1.9 计算后的内存位置

## 1.22 算术运算

大多数程序都要执行算术运算。图 1.10 对算术操作符进行了总结,注意这里使用了代数中没有的许多特殊字符。其中,星号(`*`)表示乘法,百分号(`%`)表示求模操作符,参见稍后的讨论。图 1.10 总结的算术操作符均是二元操作符,亦即要取得两个操作数的操作符。例如在 `integer1 + integer2` 中,包含了二元操作符 `+`,以及两个操作数:`integer1` 和 `integer2`。

C++ 运算	算术操作符	代数表达式	C++ 表达式
加	<code>+</code>	$f + 7$	<code>f + 7</code>
减	<code>-</code>	$p - c$	<code>p - c</code>
乘	<code>*</code>	$bm$	<code>b * m</code>
除	<code>/</code>	$x / y$ , 或 $x \div y$	<code>x / y</code>
求模	<code>%</code>	$r \bmod s$	<code>r % s</code>

图 1.10 算术操作符

整除(除数和被除数都是整数)会产生一个整数结果;例如, `7 / 4` 这个表达式的结果为 1,而 `17 / 5` 这个表达式的结果为 3。注意整除中的任何分数部分都会被简单地抛弃(或者说截掉)——不会进行四舍五入运算。

C++ 提供了求模操作符`%`,它可得到整除后的余数。求模操作符只能用于整数形式的操作数。`x % y`这个表达式会得到`x`被`y`除之后的余数。因此,`7 % 4`等于3,而`17 % 5`等于2。在以后的章节里,我们还会讨论求模操作符的许多有趣的应用,比如判断一个数字是否另一个数字的倍数(它的一个特例便是判断一个数字是奇数还是偶数)。

**常见编程错误 1.4** 试图对非整数操作数使用求模操作符`%`,会产生语法错误。

C++ 中的算术表达式必须采用直线形式输入。因此,像“`a` 被 `b` 除”这样的表达式必须写成 `a / b`,使所有常量、变量和操作符出现在一条直线中。表达式

$$\frac{a}{b}$$

对于编译器来说通常是无法接受的——尽管一些特殊用途的软件包允许以这种更自然的记号法来表示复杂的数学表达式。

在C++表达式中,括号的使用同在代数表达式中大致相同。例如,要将`a`乘以`b + c`的和,可写为

$$a * (b + c)$$

C++ 根据由下述操作符优先级规则所规定的一个精确的顺序,在算术表达式中应用不同的操作符。这些规则与代数运算规则大致相同的:

(1) 括号中的表达式的操作数先求值。所以,括号用于按程序员的意愿,强行按任何顺序进行求值。我们认为括号具有“最高的优先级”。在嵌套或嵌入括号的情况下,由内层向外层求值。

(2) 接着进行乘法、除法和求模运算。如一个表达式中包含了几个乘、除和求模运算,那么操作符按从左到右的顺序求值。我们认为乘、除和求模处在相同的优先级上。

(3) 加和减运算最后进行。如果一个表达式同时包含了几个加法和减法运算,那么操作符按从左到右的顺序应用。加和减也处于相同的优先级。

通过操作符优先级规则,C++ 可按正确的顺序应用操作符。当我们说特定的操作符从左到右应用时,实际说的是操作符的结合性。比如表达式

$$a + b * c$$

中,加法操作符`(+)`按从左到右的顺序进行结合。有的操作符还会按从右到左的顺序结合。图 1.11 总结了这些操作符优先级规则。随着附加的C++ 操作符的引入,该表还可继续扩充。本书附录 A 提供了一个完整的优先级表。

操作符	运算	求值顺序
<code>( )</code>	括号	最先求值。如括号是嵌套的,那么最内层的表达式先求值。如果同时有几对括号“处在同一级”(亦即没有嵌套),则按从左到右的顺序求值
<code>*</code> , <code>/</code> 或 <code>%</code>	乘 除 求模	其次求值。如同时有几个,则按从左到右的顺序求值
<code>+</code> 或 <code>-</code>	加 减	最后求值。如同时有几个,则按从左到右的顺序求值

图 1.11 算术操作符优先级

现在,我们参照操作符优先级规则,考虑几个不同的表达式。每个例子都列出了一个代数表达式,及对应的C++ 表达式。

下面是求5个值的算术平均值的例子:

$$\text{代数: } m = \frac{a+b+c+d+e}{5}$$

C++: `m = ( a + b + c + d + e ) / 5;`

注意括号是必需的,因为除法拥有比加法更高的优先级。要把整个和( `a + b + c + d + e` )除以5。如遗漏了括号,得到的就是 `a + b + c + d + e / 5`。它对应的代数表达式

$$a+b+c+d+\frac{e}{5}$$

是错误的。下面是一个直线形式的等式的例子:

代数: `y = mx + b`

C++: `y = m * x + b;`

无需任何括号。首先会执行乘法,因为乘法具有比加法更高的优先级。

下例包含了求模(`%`)、乘法、除法、加法和减法运算:

代数: `z = pr%q + w/x - y`

C++: `z = p * r % q + w / x - y;`

⑥ ① ② ④ ③ ⑤

其中,圆圈内的数字指出C++ 执行运算的顺序。首先,乘法、求模和除法操作符按从左到右的顺序依次执行(也就是说,这些操作符将按从左至右的方向结合起来),这是由于它们的优先级要高于加法和减法运算。接下来执行的便是加法和减法操作符——也按从左到右的顺序。最后执行的赋值操作符(=)。

注意在某些情况下,尽管表达式内包括了几对括号,但这些括号并不一定是嵌套的,就像表达式

`a * ( b + c ) + c * ( d + e )`

一样,并未包括任何嵌套括号,我们认为所有括号都“处在同一级上”。

为更好地理解操作符的优先顺序,请思考二次多项式

`y = a * x * x + b * x + c;`

⑥ ① ② ④ ③ ⑤

它的求值顺序是什么。其中,圆圈内的数字标出了C++ 执行运算时的顺序。在C++ 中,没有相应的算术操作符来表示指数,所以我们将  $x^2$  表示成 `x * x`。不久便要讨论标准库函数 `pow` (代表“power”,即指数),它可直接用于执行指数运算。由于针对 `pow` 所要求的数据类型,还存在一些小问题需要解决,所以我们将第3章才开始详细讲解 `pow`。

假定变量 `a`, `b`, `c` 和 `x` 像下面这样初始化: `a = 2`, `b = 3`, `c = 7` 和 `x = 5`。图 1.12 展示了在上述二次多项式中,操作符的应用顺序。

可在前述赋值语句中加上括号(尽管实际运算时并不需要),如

`y = ( a * x * x ) + ( b * x ) + c;`

对顺序进行澄清。

**良好编程习惯 1.15** 与代数运算一样,可在表达式中加上多余的括号,使其更清晰。这些

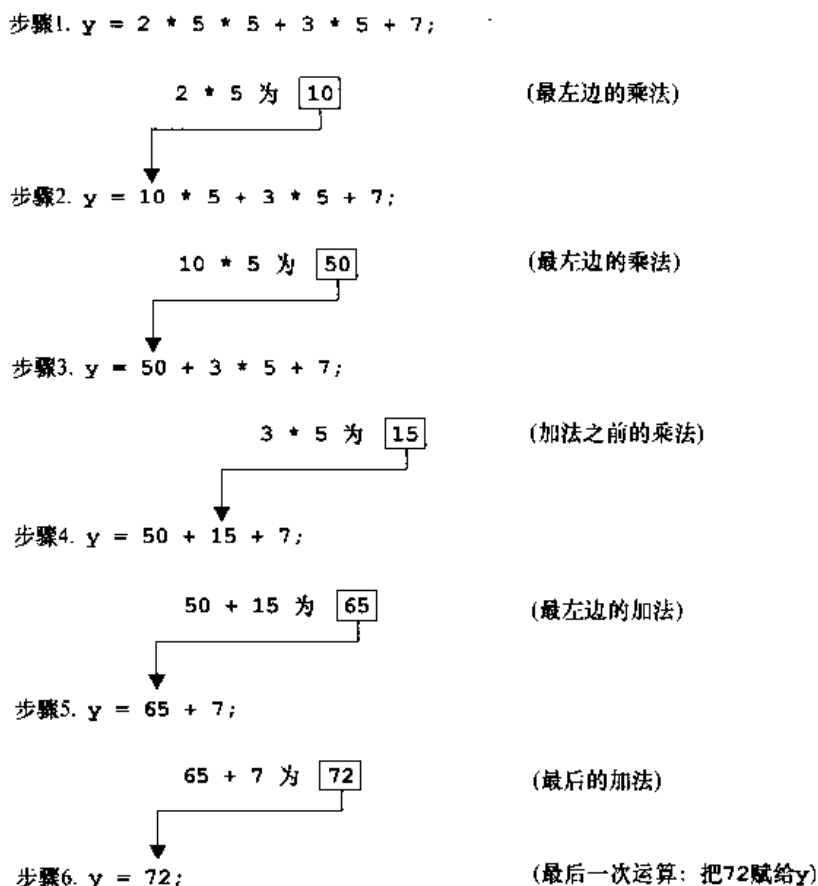


图 1.12 二次多项式求值顺序

括号叫做冗余括号。冗余括号通常用于组合大型表达式中的各个子表达式,使表达式更加清晰。将一条大型语句分割为一系列较短的、较简单的语句,叫做澄清。

## 1.23 判断:相等性和关系操作符

本节介绍C++的一个简单版本的 if 结构,它允许程序根据某些条件的真假做出决定。如条件符合,即条件为真(true),会执行 if 结构主体语句。如条件不符合,即条件为假(false),不会执行 if 结构主体语句。稍后会有一个例子对此进行说明。

if 结构中的条件可使用如图 1.13 所示的相等性操作符和关系操作符进行构建。关系操作符的优先级全部相同,而且按从左到右的顺序结合。两个相等性操作符具有相同的优先级,但要比关系操作符的优先级低。相等操作符也按从左到右的顺序结合。

**常见编程错误 1.5** 如果 `==`, `!=`, `>=` 和 `<=` 操作符的符号对之间出现空格,会出现语法错误。

**常见编程错误 1.6** 对 `!=`, `>=` 和 `<=` 这 3 个操作符来说,假如两个字符的顺序搞反了(各自写成 `!=`, `=>` 和 `=<`),便会产生语法错误。某些情况下,将 `!=` 写成 `=!` 虽然不会报告语法错误,但可以肯定是逻辑错误。

标准代数相等性操作符或关系操作符	C++ 相等性或关系操作符	C++ 条件示例	C++ 条件的含义
关系操作符			
>	>	$x > y$	x 大于 y
<	<	$x < y$	x 小于 y
$\geq$	$\geq$	$x \geq y$	x 大于或等于 y
$\leq$	$\leq$	$x \leq y$	x 小于或等于 y
相等性操作符			
=	==	$x == y$	x 等于 y
$\neq$	!=	$x != y$	x 不等于 y

图 1.13 相等性和关系操作符

**常见编程错误 1.7** 切不可将相等性操作符“==”同赋值操作符“=”混为一谈。其实按正統的逻辑,在读的时候,相等性操作符才应读成“…等于…”。相反,赋值操作符应该读成“…获得…”、“获得…的值”或者“被赋予值…”。有人喜欢把相等性操作符读成“双等于”。正如稍后所述,若混淆了这两个操作符,不仅可能出现不易辨别的语法错误,在逻辑上有时也会出现不易察觉的错误。

下面的例子用 6 个 if 结构比较用户输入的两个数字。只要满足任何一个 if 结构的条件,便会执行与 if 对应的输出语句。图 1.14 展示了该程序,以及 3 次示范执行时的输入/输出对话情况。

```

1 //Fig. 1.14: fig01_14.cpp
2 //Using if statements, relational
3 //operators, and equality operators
4 #include <iostream>
5
6 using std::cout; //program uses cout
7 using std::cin; //program uses cin
8 using std::endl; //program uses endl
9
10 int main()
11 {
12     int num1, num2;
13
14     cout << "Enter two integers, and I will tell you "
15         << "the relationships they satisfy: ";
16     cin >> num1 >> num2; //read two integers
17
18     if ( num1 == num2 )
19         cout << num1 << " is equal to " << num2 << endl;
20
21     if ( num1 != num2 )
22         cout << num1 << " is not equal to " << num2 << endl;
23
24     if ( num1 < num2 )
25         cout << num1 << " is less than " << num2 << endl;
26

```



```
27     if ( num1 > num2 )
28         cout << num1 << " is greater than " << num2 << endl;
29
30     if ( num1 <= num2 )
31         cout << num1 << " is less than or equal to "
32             << num2 << endl;
33
34     if ( num1 >= num2 )
35         cout << num1 << " is greater than or equal to "
36             << num2 << endl;
37
38     return 0; //indicate that program ended successfully
39 }
```

输出结果:

```
Enter two integers, and I will tell you
the relationships they satisfy:3 7
3 is not equal to 7
3 is less than 7
3 is lessthan or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy:22 12
22 is not equal to 12
22 us greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy:7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

图 1.14 使用相等性和关系操作符

### 第 6~8 行

```
using std::cout; //program uses cout
using std::cin; //program uses cin
using std::endl; //program uses endl
```

是 using 语句,帮助我们避免重复 std::前缀。一旦我们包含了这些 using 语句,那么可分别直接写 cout,不用写 std::cout;直接写 cin,不用写 std::cin;直接写 endl,而不用写 std::endl (注意:从此处开始,每个例子都将包含一条或多条 using 语句)。

### 第 12 行

```
int num1, num2;
```

声明了程序中使用的变量。大家应记住,变量既可在一个声明中声明,也可用多个声明进行声明。如果在一个声明中声明了多个名字(就像本例这样),就要用逗号(,)将其隔开。我们将其称为逗号分隔列表。

程序采用连续使用流读取操作(第 16 行)输入两个整数。记住,由于第 7 行的存在,所

以我们能直接写 `cin` (而不是 `std::cin`)。首先一个值读入变量 `num1` 中,然后一个值读入变量 `num2` 中。

第18行和第19行的 `if` 结构

```
if ( num1 == num2 )
    cout << num1 << " is equal to " << num2 << endl;
```

用于比较变量 `num1` 和 `num2` 的值,检测其相等性。假如值是相等的,那么第19行的语句会显示一行文本,指出数字相等。如果在第21,24,27,30和34行起始处的一条或多条 `if` 语句中,条件是真(`true`),那么对应的 `cout` 语句会显示一行文本。

注意在图1.14中,每个 `if` 结构都在它的主体中有单独一个语句,而且每个主体都进行了缩进处理。通过对 `if` 结构主体进行缩进,有助于改善程序的可读性。在第2章,我们展示了如何用多语句主体来指定 `if` 结构(将主体语句封装到一对花括号`{}`中)。

**良好编程习惯 1.16** 对 `if` 结构的主体语句采用缩进,可突出结构主体,并增强程序可读性。

**良好编程习惯 1.17** 在一个程序中,每一行只应有一条语句。

**常见编程错误 1.8** 在 `if` 结构的条件之后,假如紧接在右侧的圆括号之后放置一个分号,会造成一处逻辑错误(尽管不是语法错误)。分号会造成 `if` 结构的主体变成空的,因此 `if` 结构本身不会采取任何行动,无论它的条件是否为真。更糟的是, `if` 结构的原始主体语句现在会变成紧接在 `if` 结构之后的一条顺序执行的语句,无论如何都会执行,这通常会导致程序产生不正确的结果。

在图1.14中,请注意空白间距的使用。在C++语句中,空白字符(比如制表位、换行符和空格)通常会被编译器忽略。因此,按程序员的意愿,语句可分隔到多行上,而且可加上任意的空距。但是,假如将标识符、字符串(比如“`hello`”)和常量(比如数字1000)分隔到多行是不正确的。

**常见编程错误 1.9** 如通过插入空白字符的办法来分割一个标识符,会产生语法错误(比如将 `main` 误写为 `ma in`)。

**良好编程习惯 1.18** 较长的语句可分割到几个行上。如必须像这样分割一条语句,请挑选最合适的断点。比如对一个用逗号分隔的列表来说,可选择在某个逗号之后断开;对于较长的表达式,可考虑在一个操作符之后断开等等。一个语句分割成多行后,除第一行之外,其他所有行都采用缩进处理。

图1.15展示了本章介绍的操作符的优先级。操作符按从上升到下降序排列。注意除了赋值操作符“`=`”之外,所有这些操作符都按从左到右的顺序进行结合。加法是从左到右结合的,所以像 `x + y + z` 这样的表达式完全可以写成 `( x + y ) + z`。赋值操作符“`=`”按从右到左的顺序结合,所以像 `x = y = 0` 这样的表达式完全可以写成 `x = ( y = 0 )`。如我们马上便要看到的那样,它首先把0赋给 `y`,然后把赋值的结果(0)赋给 `x`。

**良好编程习惯 1.19** 如果一个表达式里需要包含多个操作符,请务必参考这张操作符优先级表格,核实表达式中的操作符按自己希望的顺序执行。如表达式过于复杂,以至于无法确定顺序,不妨将表达式分割为几个小语句,或者干脆用括号强行规定顺序——这样的做法和

在代数中无异。其间,请留意某些操作符(比如赋值操作符“=”)是按从右到左的顺序结合的,而非从左到右。

操作符	结合性	类型
( )	从左到右	括号
* / %	从左到右	乘
+ -	从左到右	加
<< >>	从左到右	流插入/读取
< <= > >=	从左到右	关系
= !=	从左到右	相等性
=	从右到左	赋值

图 1.15 已讨论过的所有操作符的优先级和结合性

至此,我们介绍了C++的许多重要的特性,包括在屏幕上打印数据,从键盘输入数据、执行计算和做出决定等等。第2章介绍结构化编程时,会以这些技术为基础。届时,你会越来越习惯使用缩进。另外,我们还要介绍如何指定和变化语句的执行顺序(即控制流程)。

## 1.24 对象思想:对象技术及 UML 简介

现在,我们要讨论面向对象思想。通过本节的学习,你会了解到何谓“面向对象”,面向对象其实就是在思考这个世界以及编写计算机程序的一种颇为自然的方式。

在本书前5章,我们都会把重点集中在结构化编程的“泛型”方法论上,因为我们即将构建的对象会采用结构化程序的一些组件进行合成。在每一章末尾,都有一个“对象思想”小节,它们将循序渐进地向你介绍面向对象的理论。通过这些“对象思想”小节,我们的目标是帮助你建立一种面向对象的思维方式。有了这些铺垫,即可开始应用第6章开始学习的面向对象编程知识。我们还会介绍UML(统一建模语言)。UML是一种图形化语言,允许系统构建人员(软件结构设计师、系统工程师、程序员等等)采用一种标准的记号法,对他们的面向对象设计进行呈现。

在这个必读小节(1.24节)里,介绍了基本的概念(即“对象思想”)和术语(即“对象陈述”)。在第2~5章末尾选读的“对象思想”小节,我们将采用面向对象设计(OOD)的技术来攻克一个富有挑战性的问题,并在此过程中循序渐进地探讨一些更为实际的问题。我们将分析一个典型的问题陈述,它要求构建一个系统,判断需要用哪些对象来实现该系统,判断对象需要具有哪些属性,判断这些对象需要表现的行为,以及指定对象间如何交互,以满足最终的系统要求。甚至在我们开始学习如何编写面向对象的C++程序之前,便要完成所有这些工作。在第6章、第7章和第9章末尾选读的“对象思想”小节,我们将讨论面向对象系统的一个C++实现,它的设计已在本书早期的章节中全部完成。

案例分析帮助你为即将在产业界实际遇到的各种项目做好准备。如果你是一名学生,而且你的老师不打算在课程中介绍这个案例分析,那么请考虑自学。我们相信,花时间走完这个大型和富有挑战性的项目,会对你未来的课业及职业产生极大的帮助。你会在用UML进行面向对象设计方面,打下一个扎实的基础。而且通过对一个精心编写、编制了良好文

档、而且有1 000行以上的C++程序进行彻底剖析,你的代码阅读技能将会有显著的提高。这个程序解决了案例分析中提出的各个问题。

首先从一些关键的面向对象术语开始我们的“面向对象”之旅。观察一下你周遭的现实世界,对象是无处不在的!人、动物、植物、汽车、飞机、建筑物、计算机等等……它们都是对象。人们通过对象来思考问题。我们每个人都具有非凡的抽象能力,可将屏幕上显示的一幅图像看成是人、飞机、树和山这样的对象,而非看成是单独的颜色点。如果愿意,我们可以看到沙滩而不是沙子,看到森林而不是树木,看到房子而不是砖块。

我们可能习惯于将对象划分为两个类别——动物和静物。其中,动物在某种程度上是“有生命的”。它们四处运动,同时做一些事情。而对于静物,比如毛巾等,则表面上什么事情也不做。它们只是某种静态的物品。然而,所有这些对象都有一些东西是共通的。它们都有自己的一系列属性,比如大小、形状、颜色、重量等等。而且它们都在表现出某种行为(例如,一个球会滚动、弹跳、膨胀和收缩;一个婴儿会哭、睡、爬、走和眨眼睛;一辆车会加速、刹车和转弯;而一张毛巾会吸水;等等)。

人类通过研究其属性和观察其行为,从而了解对象。不同的对象可能有相同的属性,并可表现出类似的行为。例如,你可自己比较一下婴儿和成人,人和猩猩等等。对于轿车、卡车、小童车和溜冰鞋等等来说,它们其实也有许多共通之处。

面向对象编程(OOP)以软件的形式,对现实世界的对象进行建模。它利用了类关系;在这种关系中,特定类(比如一个车辆类)的所有对象都有相同的特征。它还利用了继承关系,甚至利用了多重继承关系;在这种关系中,新创建的对象类是通过吸收现有类的特征而建立起来的,同时可能添加了自己独特的一些特征。“敞篷车”类的一个对象肯定具有一个更常规的“汽车”类的特征;但是,敞篷车的车篷还可展开和折叠,这一特征是常规的汽车所不具备的。

面向对象的编程还为我们提供了一种更加自然和直观的方式来看待编程过程。换言之,编程过程就是对现实对象及其属性/行为进行建模的一种过程。OOP还可为对象之间的通信进行建模。如同人们向同伴发送消息一样(例如军官命士兵立正),对象之间也通过消息进行沟通。

OOP将数据(属性)和函数(行为)封装到一种名为对象的包中;一个对象的数据和函数紧密地联系在一起。对象具有信息隐藏这一属性。也就是说,尽管一个对象可能知道如何通过使用良好定义的接口同另一个对象通信,但一个对象通常不允许知道其他对象内部是如何实现的——实现的细节被隐藏在对象的内部。这是非常合理的,我们完全能够很好地驾驶一部汽车,而不必知道引擎、传送机构和排气系统的内部工作原理。以后,我们会解释为了保障“良好的软件工程”,信息隐藏为何如此重要。

在C和其他过程式编程语言中,编程通常都是面向行动的;而在C++中,编程通常都是面向对象的。在C中,基本的编程单元是函数;而在C++中,基本编程单元是类,对象通过它进行实例化(即“创建”的美称)。C++类包含了函数(用于实现类的行为)和数据(用于实现类的属性)。

C程序员把重点放在函数的编写上。用于执行一些常规任务的行动可采用函数的形式进行分组,不同的函数组合到一起,即构成程序。在C中,数据理所当然是非常重要的,但数

据的主要用途便是为函数所采取的行动提供支持。在一份系统规格说明中,那些动词帮助 C 程序员决定并设计一系列协同工作的函数,它们用于实现整个系统。

C++ 程序员则把重点放在创建他们自己的用户自定义类型上,名为类和组件。每个类都包含了数据以及一系列函数,函数用于对数据进行操纵。一个类的数据组件叫做数据成员;一个类的函数组件叫做成员函数(在其他面向对象编程语言中——比如 Java——通常叫做方法)。如同内建类型(比如 int)的一个实例叫做变量一样,用户自定义类型(即“类”)的一个实例则叫做对象。程序员采用内建类型作为“基本构建单元”,以它们为基础构建自己的用户自定义类型。在 C++ 中,人们将重点放在类上(我们利用它创建对象),而不是放在函数上。在一份系统规格说明中,那些名词帮助 C++ 程序员决定并设计一系列类,以便通过它们创建一系列协同工作的对象,这些对象用于实现整个系统。

类之于对象,如同蓝图之于房屋。我们可根据一份蓝图建造许多房屋,也可根据一个类实例化(创建)许多对象。一个类可与其他类建立关系。例如,假定一家银行采取了面向对象的设计,那么 BankTeller(出纳员)类需要同 Customer(客户)类建立关系。这些关系叫做关联。

我们以后会体验到,一旦软件被打包成类,这些类便可在未来的软件系统中重复使用或重用。一组相关的类通常打包成可重复使用的组件。房地产经纪人会告诉他们的客户,影响房地产价格的 3 大因素是“地段,地段,还是地段”。类似地,我们认为会对软件开发的未来产生影响的 3 大因素是“重用,重用,还是重用”。

事实上,利用对象技术,今后大多数软件都可通过合并一系列“标准化、可互换的零件”构建,这些零件的名字叫做“类”。本书将教你如何“改良宝贵的类”,以达到重用、重用、再重用之目的。你创建的每个类都有可能成为一项价值无限的软件资产,你和其他程序员可用它加快未来软件开发的速度,并改进其质量。类的重用,大有潜力可挖。

### 1.24.1 面向对象分析和设计(OOAD)概述

迄今为止,你可能已用 C++ 写过几个程序。那么,你是怎样为自己的程序创建代码的?如果你和其他许多刚入门的程序员一样,可能会打开计算机,然后简单地开始键入代码。这个办法对小项目来说是可行的,但假如有人要求你创建一个大的软件系统,比如对一家大银行的自动取款机进行控制,又该怎么做呢?像这样的一个项目如此庞大而复杂,绝不可能简单地坐下来,然后逐行敲程序了事。

为创建最佳方案,应按一个详细的软件开发过程行事。也就是说,针对具体的项目需求,先对其进行分析;然后开发出一个设计,以满足那些需求。逐步骤完成这一过程,然后把结果拿给上级审批,最后才能开始为自己的项目编写代码。如这个过程需要从面向对象的角度来分析和设计系统,就可以把它叫做面向对象分析和设计(OOAD)过程。有经验的程序员知道,一个系统开发方案事先没有计划好,往往会半途而废,浪费大量时间。相反的,表面上似乎很复杂的问题,只要事先进行精心的分析和设计,便可事半功倍。

我们需要先分析一个问题,再开发一个方案来解决它——整个过程背后的基本思想可用 OOAD 这一常规术语来表示。对一些小问题来讲,比如前几章的那些,它们并不要求牵涉十分复杂的分析过程。对于这样的问题,通常只需先写好伪代码,然后就可开始写代码了

(伪代码是对程序代码进行表达的一种非正式手段。它实际并不是程序语言,但我们可把它作为某种“提纲”使用,以便在写代码时提供辅助。第2章将详细探讨伪代码的问题)。

伪代码对小问题来说也许已足够,但随着问题以及解决这些问题的人数的增加,OOAD的方法会显得更加有效。理想情况下,一个开发小组应遵守一个经严格定义的、用于解决问题的过程。同时,还要采用某种统一的方式,相互间就那个过程的各种结果进行沟通。目前有多种不同的OOAD过程可供选择;但是,可对任意OOAD过程的结果进行沟通的一种图形化语言正在得到人们的广泛使用。这一语言便是UML(统一建模语言)。UML是在20世纪90年代中期开发成功的,最初有3位软件理论家提供指导:Grady Booch,James Rumbaugh 和 Ivar Jacobson。

### 1.24.2 UML 的历史

20世纪80年代,越来越多的单位开始用OOP编写程序。人们越来越迫切地需要一种成熟的过程来实现OOAD的方法。许多理论家(包括Booch,Rumbaugh和Jacobson)都从个人的角度,开发并宣传一些独立的过程,以满足这方面的需要。每个类似过程都有各自的记号法,或称“语言”(采用示意图的形式),以表达分析和设计结果。

20世纪90年代初,不同的公司(甚至同一家公司的不同部门)都在采用不同的过程和记号法。另外,这些公司希望使用软件工具来支持它们的特定的过程。由于存在如此多的过程,所以软件厂商觉得很难提供这样的工具。显然,制定一套标准化的过程和记号法势在必行。

1994年,James Rumbaugh与Rational Software公司的Grady Booch合作,两个人着手统一各自颇受欢迎的开发过程。不久,Ivar Jacobson也加入这个小组。1996年,他们面向软件工程社区发布了UML的早期版本,并请求人们提供反馈意见。与此同时,一家名为对象管理组(OMG)的组织向社会各界征求一种通用建模语言的方案。OMG是一个非赢利性组织,主要宗旨是针对面向对象的技术,发布一系列准则和规范,从而宣传及促进面向对象技术的应用。当时,已有几家公司——包括HP,IBM,Microsoft和Rational Software——认识到了人们对一种通用建模语言的迫切需求。这些公司联合成立了UML Partners协会,以响应OMG的邀请。协会开发并提交了UML版本1.1,并提交给OMG。OMG接受了这一提议,并于1997年,宣布由自己负责UML的后续维护及修订工作。1999年,OMG发布了1.3版UML(本书出版时的最新版本UML 1.5)。

### 1.24.3 什么是UML

针对面向对象系统的建模,UML(统一建模语言)目前是应用得最广的图形化表示方案。它实现了当初设计者的目标,真的将20世纪80年代末流行的各种记号方案统一起来了。凡是设计系统的人,都可使用这一语言(采用示意图的形式)对系统进行建模。

UML最吸引人的特性之一,便是它强大的灵活性。UML是可扩展的,且独立于许多具体的OOAD过程。现在,UML建模人员完全能采用各种过程来开发自己的系统,但所有开发者现在都可用一套标准的记号法来表示那些系统。

UML是一种复杂的、具有丰富特性的图形化语言。在本书的各个“对象思想”小节里,

我们展示了这些特性的一个简化的子集。然后再用这个子集来指导读者获得初次的 UML 设计体验。当然,这些内容面向的是那些刚入门的面向对象设计者/程序员。欲阅读有关 UML 的一个更完整的讨论,请参考 OMG 的 Web 站点([www.omg.org](http://www.omg.org)),并可阅读 UML 1.3 正式规范文档([www.omg.org/uml/](http://www.omg.org/uml/))。目前已出版了许多有关 UML 的书籍。其中,由 Martin Fowler 与 Kendall Scott 共同编著的《UML Distilled(第二版)》对 UML 1.3 进行了详细介绍,同时附带大量实例。另外,由 Booch, Rumbaugh 和 Jacobson 共同编著的《The Unified Modeling Language User Guide》无疑是最权威的 UML 教程。

面向对象技术目前已成为软件业最流行的话题,UML 的情况也差不多。在本书的各个“对象思想”小节里,我们的目标是鼓励你尽早、尽量多用面向对象的方式进行思考。从第 2 章末的“对象思想”一节开始,你将开始运用对象技术来实现一个方案,以解决一个真正的问题。希望这个可选项目能成为你的引路人,让你愉快迎接挑战,学习用 UML 进行面向对象的设计,以及最终进行面向对象的编程。

## 1.25 小结

- 所谓计算机,是指能执行计算并做出逻辑决策的一种设备,其运算速度是人脑的数百万乃至数十亿倍。
- 计算机在计算机程序的控制下处理数据。
- 计算机由多种设备构成,比如键盘、屏幕、磁盘、内存和处理单元等等,我们把这些设备统称为硬件。
- 在计算机上运行的计算机程序称为软件。
- 输入单元是计算机的“接收”区域。今天,大多数信息都是通过类似于打字机的键盘输入到计算机中的。
- 输出单元是计算机的“发送”区域。如今计算机输出的大多数内容都直接显示在屏幕上,或者打印在纸上。
- 内存单元是计算机的“仓库”区域,通常叫做内存或主存储器。
- 算术和逻辑单元负责执行计算和做出决策。
- 一些不常被其他单元用到的程序或数据通常放在辅助存储设备里(比如磁盘),直至再次需要它们。
- 在单用户批处理中,计算机一次执行一个程序,同时以组或成批的方式来处理数据。
- 操作系统是一种软件系统,可更有效地使用计算机,并可从计算机获得最佳性能。
- 多路编程操作系统允许在计算机上“同时”进行多项工作——计算机在不同的工作之间共享其资源。
- 时间共享是多路程序的一种特殊情况。在其中,用户要通过终端来访问计算机。用户的程序表面上是同时运行的。
- 采用分布式计算方式,单位内部的计算工作将通过网络分布到不同站点中进行。
- 服务器负责存储程序和数据,它们可由分布在整个网络中的客户计算机共享。这正是“客户机/服务器计算”这一术语的来历。

- 任何计算机都能直接理解它自己的机器语言。在机器语言中,通常包括大量数值串(最终可简化成一系列的1和0),它们指示计算机每次执行它的一个最基本的操作。机器语言与机器有关。
- 英语风格的缩写构成了汇编语言的基础。汇编程序负责将汇编语言翻译成机器语言。
- 编译器可将高级语言翻译成机器语言。高级语言包含一些英语词汇以及常用数学符号。
- 解释器程序直接执行高级语言程序,不需要将那些程序编译成机器语言。
- 尽管编译好的程序执行起来要比解释过的程序快,但在程序开发环境中,最流行的还是解释器程序。在这种环境中,随着新功能的引入,以及错误的不断纠正,要求程序经常发生改变。一个程序开发好之后,就可生成它的一个编译版本,以便更有效地运行。
- 用C和C++编写的程序可移植到大多数计算机。
- FORTRAN(FORmula TRANslator,公式翻译器)语言用于数学应用。COBOL(COMmon Business Oriented Language,标准商务语言)主要用于商业应用(这些应用要求精确和高效地处理大量数据)。
- 结构化编程是一种比较严格的程序编写机制,用它可方便地写出思路比非结构化程序清晰、更易测试和调试以及更易修改的程序。
- Pascal设计用于在学术环境中进行结构化编程的教学。
- Ada是在美国国防部赞助下开发出来的一种语言,以Pascal作为基础。
- 多任务允许程序员指定并行的操作。
- 所有C++系统都由3个部件构成:环境、语言 and 标准库。库函数不属于是C++;它们可执行数学计算这样的常用操作。
- C++程序通常要经历6个执行阶段,其中包括:编辑、预处理、编译、连接、装入和执行。
- 程序员用编辑器输入C++程序。如有必要,还要纠正其中的错误。在一个典型的UNIX系统上,C++文件的名称以.C扩展名结尾。
- 编译器负责将C++程序翻译成机器语言码(或目标码)。
- C++预编译程序将遵从一些名为预处理程序指令的特殊命令,它们通常用于指定要包括到打算编译的文件中的文件,以及指定要用程序文本替换的特殊符号等等。
- 连接器将目标码同缺失函数的代码连接起来,以生成一个可执行映像(其中不再有缺失的部分)。在典型的UNIX系统中,用于编译和连接一个C++程序的命令是CC。假如程序经过正确编译和连接,会生成一个名为a.out的文件。它是程序的可执行映像。
- 装入器从磁盘取得一个可执行映像,并将其传输至内存。
- 计算机在其CPU的控制下,以每次一条指令的形式执行程序。
- “被零除”这样的错误可能发生于程序运行时,所以这些错误叫做运行期或执行期错误。



- 被零除通常被认为是一种致命错误;也就是说,是一种会导致程序立即中止,无法成功完成其工作的一种错误。非致命错误虽然允许程序运行至结束,但通常会产生不正确的结果。
- 特定的C++ 函数会从 cin(标准输入流)读取自己的输入,最常见的 cin 便是键盘。但是, cin 也可同另一个设备连接。数据则通常输出至 cout(标准输出流),这通常就是计算机屏幕,但 cout 也可同另一个设备连接。
- 标准错误流叫做 cerr。cerr 流(通常同屏幕连接)用于显示错误消息。
- 在不同的C++ 实现和不同的计算机之间,存在着大量差异,这使得完美的移植性是一个难以实现的目标。
- C++ 具有进行面向对象编程的能力。
- 对象本质是一种可重复使用的软件组件,用于对现实世界中的各种物件进行建模。对象是根据名为“类”的一种“蓝图”而构建起来的。
- 单行注释以//开头。程序员插入注释以编制程序文档,并增强其可读性。程序运行时,注释不会导致计算机为之采取任何行动。
- #include <iostream> 告诉C++ 预编译器将输入/输出流头文件包括到程序中。只有利用该文件包含的信息,采用了 std::cin 和 std::cout 以及 << 和 >> 操作符的程序才能正常编译。
- C++ 程序首先执行的是 main 函数。
- 输出流对象 std::cout——通常连接到屏幕——用于输出数据。要同时输出多个数据项,可连接流插入(<<)操作符。
- 输入流对象 std::cin——通常连接到键盘——用于输入数据。要同时输入多个数据项,可连接流读取(>>)操作符。
- C++ 程序中的所有变量在使用前都必须声明。
- C++ 中的变量名可为任意有效标识符。一个标识符可由一系列字符构成,其中包括字母、数字和下划线(\_),但却不能以一个数字开头。C++ 标识符可为任意长度,但是,有的系统和/或具体的C++ 实现有可能对标识符的长度进行了某种限制。
- C++ 要求严格区分大小写。
- 大多数计算都是在赋值语句中执行的。
- 计算机内存中存储的每一个变量都有一个名称、一个值、一个类型和一个长度。
- 一个新值置入内存位置后,它会替换那个位置以前保存的值。以前的值会被丢失。
- 从内存读一个值时,整个过程是非破坏性的。也就是说,只会读取那个值的一个副本,原始值则原封未动。
- C++ 按一个精确的顺序对算术表达式进行求值,这个顺序由操作符的优先级规则和结合性决定。
- if 语句允许程序在符合条件的情况下做出一项决定。if 语句的格式是:

```
if ( 条件 )  
    语句;
```

如“条件”为真,则执行 if 主体中的语句。如条件不符合(即条件为假),则跳过主体语句。

- if 语句的中的条件通常采用相等性操作符和关系操作符来构成。使用这些操作符的结果肯定要么是真,要么是假。
- 下述语句

```
using std::cout;
using std::cin;
using std::endl;
```

均为 using 语句,帮助我们避免重复 std:: 前缀。一旦包含了这些 using 语句,就可直接写 cout,不用写 std::cout; 直接写 cin,不用写 std::cin; 直接写 endl,而不用写 std::endl。

- 面向对象是思考这个世界以及编写计算机程序的一种自然的方式。
- 对象不仅拥有一系列属性(大小、形状、颜色、重量等等),还可表现出一系列行为。
- 人类通过研究其属性和观察其行为,从而了解对象。
- 不同的对象可能有许多相同的属性,并可表现出类似的行为。
- 面向对象编程(OOP)以软件的形式,对现实世界的对象进行建模。它利用了类关系;在这种关系中,特定类的所有对象都有相同的特征。它还利用了继承关系,甚至利用了多重继承关系;在这种关系中,新创建的对象类是通过吸收现有类的特征而建立起来的,同时可能添加了自己独特的一些特征。
- 面向对象的编程还为我们提供了一种更加自然和直观的方式来看待编程过程。换言之,编程过程就是对现实对象及其属性/行为进行建模的一种过程。
- OOP 还可通过消息,为对象之间的通信进行建模。
- OOP 将数据(属性)和函数(行为)封装到对象中。
- 对象具有信息隐藏这一属性。尽管一个对象可能知道如何通过经过良好定义的接口同另一个对象通信,但一个对象通常不允许知道其他对象具体是如何实现的。
- 为保证良好的软件工程,信息隐藏至关重要。
- 在 C 和其他过程式编程语言中,编程通常都是面向行动的。数据在 C 中是理所当然最重要的,但数据的主要用途是为函数所采取的行动提供支持。
- C++ 程序员把重点放在创建他们自己的、名为“类”的一种用户自定义类型上。每个类都包含了数据以及一系列函数,函数用于对数据进行操纵。一个类的数据组件叫做数据成员。一个类的函数组件叫做成员函数或方法。

## 本章术语

abstraction 抽象

action 行动

action-oriented 面向行动

analysis 分析

ANSI/ISO standard C ANSI/ISO 标准 C

ANSI/ISO standard C++ ANSI/ISO 标准 C++

arithmetic and logic unit( ALU) 算术和逻辑单元

arithmetic operators 算术操作符

assembly language 汇编语言

assignment operator( = ) 赋值操作符( = )

association 关联

associativity of operators 操作符的结合性

attribute 属性

attributes of an object 对象的属性

behavior 行为

behaviors of an object 对象的行为

binary operator 二元操作符

body of a function 函数主体

- C++ standard library C++ 标准库
- case sensitive 区分大小写
- central processing unit(CPU) 中央处理单元(CPU)
- cerr object cerr 对象
- cin object cin 对象
- clarity 澄清
- class 类
- client/server computing 客户机/服务器计算
- comma-separated list 逗号分隔列表
- comment(//) 注释(//)
- compile error 编译错误
- compile-time error 编译期错误/编译时错误
- compiler 编译器
- component 组件
- computer 计算机
- computer program 计算机程序
- condition 条件
- cout object cout 对象
- crafting valuable classes 改良宝贵的类
- data 数据
- data member 数据成员
- decision 决定/决策
- declaration 声明
- design 设计
- distributed computing 分布式计算
- editor 编辑器
- encapsulation 封装
- equality operators 相等性操作符
- escape character(\) 换码字符(\)
- escape sequence 换码序列
- execution-time error 执行期错误
- fatal error 致命错误
- file server 文件服务器
- flow of control 控制流程
- function 函数
- hardware 硬件
- high-level language 高级语言
- identifier 标识符
- if structure if 结构
- information hiding 信息隐藏
- inheritance 继承
- input device 输入设备
- input/output(I/O) 输入/输出(I/O)
- instantiate 实例化
- integer(int) 整数(int)
- integer division 整除
- interface 接口
- interpreter 解释器
- left-to-right associativity 从左到右结合
- linking 连接
- loading 装入
- logic error 逻辑错误
- machine dependent 依赖于机器
- machine independent 不依赖于机器/与机器无关
- machine language 机器语言
- member function 成员函数
- memory 内存
- memory location 内存位置
- message 消息
- method 方法
- modeling 建模
- multiple inheritance 多重继承
- modulus operator(%) 求模操作符(%)
- multiplication operator(\*) 乘法操作符(\*)
- multiprocessor 多处理器
- multiprogramming 多路程序
- multitasking 多任务
- nested parentheses 嵌套括号
- newline character(\n) 换行符(\n)
- non-fatal error 非致命错误
- nouns in a system specification  
系统规格说明中的名词
- object 对象
- Object Management Group(OMG)  
对象管理组(OMG)
- object-oriented analysis and design(OOAD)  
面向对象分析和设计(OOAD)
- object-oriented design(OOD)  
面向对象设计(OOD)
- object-oriented programming(OOP)  
面向对象编程(OOP)
- operand 操作数
- operator 操作符
- operator associativity 操作符的结合性
- output device 输出设备
- parentheses 圆括号()
- precedence 优先级
- preprocessor 预处理器
- primary memory 主内存
- procedural programming 过程式编程

procedural programming language 过程式编程语言	standard input object(cin) 标准输入对象(cin)
programming language 编程语言/程序语言	standard output object(cout) 标准输出对象(cout)
prompt 提示	statement 语句
pseudocode 伪代码	statement terminator 语句中止符(;) )
relational operators 关系操作符	string 字符串
requirements 需求	structured programming 结构化编程
reserved words 保留关键字	syntax error 语法错误
reuse, reuse and reuse 重用、重用、再重用	translator program 翻译程序
right-to-left associativity 从右到左结合	Unified Modeling Language(UML)
rules of operator precedence 操作符优先级规则	UML(统一建模语言)
run-time error 运行期错误/执行期错误	user-defined type 用户自定义类型
semicolon(;) statement terminator	variable 变量
分号(;)语句中止符	variable name 变量名
software 软件	variable value 变量值
software asset 软件资产	verbs in a system specification
software reusability 软件重用性	系统规格说明中的动词
standard error object(cerr) 标准错误对象(cerr)	white-space characters 空白字符

## 常见编程错误

- 1.1 “被零除”这样的错误是在程序运行时发生的,所以这类错误叫做运行期错误或者执行期错误。被零除通常被认为是一种致命错误;也就是说,是一种会导致程序立即中止,无法成功完成其工作的错误。非致命错误则允许程序运行至结束,但通常会产生不正确的结果(注意:在某些系统上,被零除并不是致命错误。详情可参见系统文档)。
- 1.2 在需要从键盘输入数据,或者需要将数据输出至屏幕的程序中,假如忘了包括 iostream 文件,会导致编译器报告一条错误消息。
- 1.3 遗漏语句末尾的分号属于语法错误。一旦编译器不能正确识别语句,便会导致语法错误。编译器通常会发出一条错误消息,帮助程序员定位并纠正不正确的语句。语法错误是对语言规则的违背。语法错误也叫做编译错误、编译期错误或编译时错误(因为它们是在编译时显现出来的)。
- 1.4 试图对非整数操作数使用求模操作符%,会产生语法错误。
- 1.5 如果 ==、!=、>= 和 <= 操作符的符号对之间出现空格,会出现语法错误。
- 1.6 对 !=、>= 和 <= 这三个操作符来说,假如两个字符的顺序搞反了(各自写成 =!, => 和 =<),便会产生语法错误。某些情况下,将 != 写成 =! 虽然不会报告语法错误,但可以肯定是逻辑错误。
- 1.7 切不可混淆相等性操作符“==”同赋值操作符“=”混为一谈。其实按正统的逻辑,在读的时候,相等性操作符才应读成“…等于…”。相反,赋值操作符应该读成“…获得…”、“获得…的值”或者“被赋予值…”。有人喜欢把相等性操作符读成“双等于”。如果大家稍后便会看到的那样,如混淆了这两个操作符,不仅可能出现不易辨别的语法错误,在逻辑上有时也会出现不易察觉的错误。
- 1.8 在 if 结构的条件之后,假如紧接在右侧的圆括号之后放置一个分号,会造成一处逻辑错误(尽管不是语法错误)。分号会造成 if 结构的主体变成空的,因此 if 结构本身不会采

取任何行动,无论它的条件是否为真。更糟的是,if结构的原始主体语句现在会变成紧接在if结构之后的一条顺序执行的语句,无论如何都会执行,这通常会导致程序产生不正确的结果。

- 1.9 如通过插入空白字符的办法来分割标识符,会产生语法错误(比如将main误写为ma in)。

## 良好编程习惯

- 1.1 C++程序应以简单和直接的方式编写。这有时也称为KIS(“keep it simple”的简称,即“尽量简单”)编程方法。千万不要去尝试一些古怪的用法,去“折磨”这种语言。
- 1.2 仔细阅读你所用的C++版本的用户手册。经常查阅这些手册,才能确保自己能了解并正确使用C++的丰富特性。
- 1.3 计算机和编译器是最好的老师。仔细阅读了C++语言手册之后,如果你不能确定一项C++的特性是如何工作的,不妨用一个短小的“测试程序”进行试验,看其效果。设置编译器选项,令其报告“最多的警告”。研究程序编译时出现的每一条错误消息,并对程序进行纠正,去除这些消息。
- 1.4 每个程序都应以注释开头,以描述该程序的用途。
- 1.5 许多程序员让函数打印的最后一个字符是换行符(\n)。这样可保证函数将屏幕光标定位在一个新行的起始处。这样一个自发的约定可促进软件的重用能力——此为软件开发环境的一个重要目标。
- 1.6 针对每个函数的主体,令其在定义函数主体的花括号内部,缩进一个级别的位置。这样可使程序的函数结构更清晰,增强其可读性。
- 1.7 先为你喜欢的缩进距离拟出一个约定,然后始终坚持这一约定。可考虑用制表位(按Tab键)生成缩进。但是,不同的系统其制表位的距离往往不同。因此,建议你要么使用1/4英寸制表位,要么(一种更好的做法)用3个空格构成一个缩进级别。
- 1.8 有的程序员喜欢行行声明一个变量。采用这种格式,可方便地在每个声明后插入说明性的注释内容。
- 1.9 每个逗号(,)后面都应插入一个空格,以增强程序的可读性。
- 1.10 挑选一个有意义的变量名,将有助于保障程序的“自编档能力”;也就是说,只需读一读程序,即可轻松理解它,而不是必须求助于手册,或使用过多的注释。
- 1.11 避免标识符以下划线和双下划线开头,因为C++编译器可能采用这种形式的名称为其内部的某些用途提供服务。这样,有助于避免你选择的名称同编译器选择的名称混淆。
- 1.12 可执行语句之间的声明之前,需插入一个空行。这样可在程序中突出声明语句,使程序更加清晰。
- 1.13 如果你喜欢在一个函数的起始处放置声明,请用一个空行,以区分声明与函数中的可执行语句,突出声明结束的位置和可执行语句的开始位置。
- 1.14 在二元操作符的两端,请分别添加一个空格。这样可突出显示操作符,增强程序可读性。

- 1.15 与代数运算一样,可在表达式中加上多余的括号,使其更清晰。这些括号叫做冗余括号。冗余括号通常用于组合大型表达式中的各个子表达式,使表达式更加清晰。将一条大型语句分割为一系列较短的、较简单的语句,叫做澄清。
- 1.16 对if结构的主体语句采用缩进,可突出结构主体,并可增强程序可读性。
- 1.17 在一个程序中,每一行只应有一条语句。
- 1.18 较长的语句可分割到几个行上。如必须像这样分割一条语句,请挑选最合适的断点。比如对一个用逗号分隔的列表来说,可选择在某个逗号之后断开;对于较长的表达式,可考虑在一个操作符之后断开等等。一个语句分割成多行后,除第一行之外,其他所有行都进行缩进处理。
- 1.19 如果一个表达式里需要包含多个操作符,请务必参考操作符优先级表,核实表达式中的操作符按自己希望的顺序执行。如表达式过于复杂,以至于无法确定顺序,不妨将表达式分割为几个小语句,或者干脆用括号强行规定顺序——这样的做法和在代数中无异。其间,请留意某些操作符(比如赋值操作符=)是按从右到左的顺序结合的,而非从左到右。

### 性能提示

- 1.1 采用标准库的函数和类,而不是自行编写对应的版本,可有效提高程序性能,因为这些组件已经过精心的编写,已保证高效而正确地运行。
- 1.2 尽量重复使用现有代码组件,而不是去开发自己的版本,这将有助于提升程序性能,因为这些组件通常在当初编写时就考虑到了执行效率的问题。

### 可移植性提示

- 1.1 由于C是一种标准化的、与硬件无关的、广泛采用的语言,所以用C写成的应用程序通常无需修改,或者只需进行少许修改,即可用于多种不同的计算机系统。
- 1.2 采用标准库函数和类,而不是编写自己的相应版本,可显著改善程序的移植性。这是由于在几乎所有的C++实现方案中,都已包含了这些标准库函数和类。
- 1.3 尽管有可能写出可移植的程序,但在不同的C和C++编译器以及不同的计算机之间,会产生许多问题。这些问题的存在,加大了移植程序的难度。并不是说用C和C++写程序,就能保证可移植性。程序员通常需要直接同特定的编译器和计算机平台打交道。
- 1.4 C++语言本身允许任意长度的标识符,但你的系统和/或具体的C++实现有可能对标识符的长度进行了某种限制。因此,请使用31个字符以内的标识符,以保证可移植性。

### 软件工程知识

- 1.1 使用“组装方法”创建程序。尽量不重复别人已经做过的工作。尽可能利用现有的构建单元——这正是“软件重用”的主旨,也是面向对象编程的精髓所在。
- 1.2 进行C++编程时,通常要使用以下基本构建单元:来自C++标准库的类和函数、你自行创建的类和函数以及由一些著名的第三方库提供的类和函数。
- 1.3 一系列包容全面的可重用软件组件类库可通过因特网和万维网取得。类似的许多库都

是免费的。

## 自测题

### 1.1 填空题:

- a) 全球首家引入“个人计算”理念的公司是\_\_\_\_\_。
- b) 第一款被工商界广泛接纳的“个人计算机”产品是\_\_\_\_\_。
- c) 计算机在一系列指令的控制下处理数据,这一系列指令统称为计算机\_\_\_\_\_。
- d) 计算机的6大关键逻辑单元是:\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- e) 本章讨论的3类主流计算机语言是\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- f) 将用高级语言写的程序翻译成机器语言的工具叫做\_\_\_\_\_。
- g) C 以\_\_\_\_\_操作系统的开发语言而著称。
- h) 由 Wirth 开发成功的\_\_\_\_\_语言主要用于在大学里教授结构化编程的概念。
- i) 美国国防部开发出的 Ada 语言支持\_\_\_\_\_功能,它使程序员在自己的程序中,能事先规定好同时采取的数项行动。

### 1.2 完成下列有关C++ 环境的陈述:

- a) C++ 程序通常用一个\_\_\_\_\_程序输入计算机。
- b) 在一个C++ 系统中,在计算机的翻译阶段开始之前,要先执行一个\_\_\_\_\_程序。
- c) \_\_\_\_\_程序将编译器的输出同各种库函数合并起来,以生成一个可执行映像。
- d) \_\_\_\_\_程序将C++ 程序的可执行映像从磁盘传输到内存。

### 1.3 填空题:

- a) 每个C++ 程序首先从\_\_\_\_\_函数开始执行。
- b) \_\_\_\_\_用于开始每一个函数的主体,而\_\_\_\_\_用于结束每一个函数的主体。
- c) 每个语句都以一个\_\_\_\_\_结尾。
- d) 换码序列\n 代表\_\_\_\_\_符,它会导致光标定位到屏幕下一行的行首。
- e) \_\_\_\_\_语句用于做出决定。

### 1.4 判断正误。如果有错,请说明理由。假定已使用 using std::cout; 语句。

- a) 程序执行时,注释会导致计算机在屏幕上打印出//之后的文字。
- b) 如果用 cout 输出换码序列\n,会导致光标定位到屏幕下一行的起首。
- c) 所有变量在使用前都必须进行声明。
- d) 所有变量在声明时都必须规定一种类型。
- e) C++ 认为 number 和 NuMbEr 这两个变量是完全一样的。
- f) 声明可在C++ 函数主体中的几乎任何地方出现。
- g) 求模操作符(%)只能用于对整数操作数进行操作。
- h) 算术操作符 \*、/、%、+ 和 - 优先级相同。
- i) 用于打印3行输出的一个C++ 程序必须包含3个使用了 cout 的输出语句。

### 1.5 针对下列任务,分别编写C++ 语句(假定不使用 using 语句)。

- a) 将变量 c、thisIsAVariable、q76354 和 number 声明成 int 类型。

- b) 提示用户输入一个整数。输入你的提示消息后,加一个分号(:),再加一个空格,最后使光标停留在那个空格之后。
- c) 从键盘读取用户输入的一个整数,并将输入值存储到一个整数变量 `age` 中。
- d) 假如变量 `number` 不等于 7,打印“The variable number is not equal to 7”字样。
- e) 用一行打印消息“This is a C++ program”。
- f) 用两行打印消息“This is a C++ program”,第一行以C++ 结尾。
- g) 打印消息“This is a C++ program”,每个单词单独占一行。
- h) 打印消息“This is a C++ program”,每个单词之间用一个制表位分隔。

1.6 针对下列任务,分别编写C++ 语句(假定使用了 `using` 语句)。

- a) 指出一个程序将计算 3 个整数的乘积。
- b) 将变量 `x`, `y`, `z` 和 `result` 声明成 `int` 类型。
- c) 提示用户输入 3 个整数。
- d) 从键盘读取 3 个整数,并将它们分别保存到变量 `x`, `y` 和 `z` 中。
- e) 计算包含在 `x`, `y` 和 `z` 中的 3 个整数的乘积,并将结果赋给变量 `result`。
- f) 打印“The product is”字样,后面带变量 `result` 的值。
- g) 从 `main` 返回一个值,指出程序成功中止。

1.7 利用练习题 1.6 中编写的语句,写一个完整的程序,使其计算和显示 3 个整数的乘积。注意:你需要编写必要的 `using` 语句。

1.8 指出并改正下述各条语句中的错误(假定使用了 `using std::cout;` 语句):

- a) `if ( c < 7 );`  
`cout << "c is less than 7 \n";`
- b) `if ( c = > / )`  
`cout << "c is equal to or greater than 7 r.";`

1.9 填写正确的“对象”术语:

- a) 人们可以在看电视的时候看到了上面的颜色点;另外,他们也可能回过头来,看到有 3 个人坐在一张会议桌旁;这是一种名为\_\_\_\_\_的能力的例子。
- b) 如果把汽车看作一个对象,那么就“汽车能够改装”这一事实来说,它属于汽车的一种属性/行为(请选择一个)\_\_\_\_\_。
- c) 汽车可以加速或减速,可以左右转弯,也可前进或后退——诸如此类的事实,全都是一个汽车对象的\_\_\_\_\_的例子。
- d) 一个新类从几个不同的现成类继承特征时,这叫做\_\_\_\_\_继承。
- e) 对象通过相互间发送\_\_\_\_\_来进行通信。
- f) 对象相互之间通过经良好定义的\_\_\_\_\_来进行通信。
- g) 每个对象最初都不允许知道其他对象是如何实现的;这种属性叫做\_\_\_\_\_。
- h) 在系统规格说明中,\_\_\_\_\_可帮助C++ 程序员判断为实现系统需要哪些类。
- i) 一个类的数据组件叫做\_\_\_\_\_,而一个类的函数组件叫做\_\_\_\_\_。
- j) 用户自定义类型的一个实例叫做一个\_\_\_\_\_。



## 自测题答案

- 1.1 a) 苹果计算机公司 b) IBM 个人计算机 c) 程序 d) 输入单元、输出单元、内存单元、算术和逻辑单元(ALU)、中央处理单元(CPU)和辅助存储单元 e) 机器语言、汇编语言 and 高级语言 f) 编译器 h) UNIX i) Pascal j) 多任务
- 1.2 a) 编辑器 b) 预处理器 c) 连接器 d) 装入器
- 1.3 a) main b) 左花括号({), 右花括号(}) c) 分号 d) 换行 e) if
- 1.4 a) 错误。程序执行时, 注释不会导致计算机采取任何行动。它们用于为程序编档, 并改善其可读性。  
 b) 正确。  
 c) 正确。  
 d) 正确。  
 e) 错误。C++ 要区分大小写, 所以这两个变量并不相同。  
 f) 正确。  
 g) 正确。  
 h) 错误。\*、/和%操作符优先级相同, 但+和-操作符优先级较低。  
 i) 错误。用一条使用了cout的输出语句, 只需在其中包含多个\n换码序列, 即可同时打印多行。
- 1.5 a) `int c, thisIsAVariable, q76354, number;`  
 b) `std::cout << "Enter an integer:";`  
 c) `std::cin >> age;`  
 d) `if ( number != 7)`  
     `std::cout << "The variable number is not equal to 7 \n";`  
 e) `std::cout << "This is a C++ program\n";`  
 f) `std::cout << "This is a C++ \nprogram\n";`  
 g) `std::cout << "This \nis \na \nC++ \nprogram\n";`  
 h) `std::cout << "This \tis \ta \tC++ \tprogram\n";`
- 1.6 a) `//Calculate the product of three integers`  
 b) `int x, y, z, result;`  
 c) `cout << "Enter three integers:";`  
 d) `cin >> x >> y >> z;`  
 e) `result = x * y * z;`  
 f) `cout << "The product is" << result << endl;`  
 g) `return 0;`
- 1.7 `//Calculate the product of three integers`  
`#include <iostream>`  
  
`using std::cout;`  
`using std::cin;`

```
using std::endl;

int main ()
{
    int x, y, z, result;

    cout << "Enter three integers:";
    result = x*y*z;
    cout << "The product is" << result << endl;

    return 0;
}
```

1.8 a) 错误:if 语句条件部分的右括号之后,多了一个分号。

改正:删除右括号之后的分号。

注意:由于这个错误,造成不管 if 语句的条件是否为“true”,输出语句都会执行。右括号后的分号被当作一个空语句对待——亦即什么事情都不干的语句。我们将在下一章学习有关空语句的更多知识。

b) 错误:条件操作符 ==>

改正:把 ==> 改为 >=。

1.9 a) 抽象 b) 属性 c) 行为 d) 多重 e) 消息 f) 接口 g) 信息隐藏

h) 名词 i) 数据成员;成员函数或方法 j) 对象

## 练习题

1.10 按硬件和软件对下列项目进行分类:

- |         |             |
|---------|-------------|
| a) CPU  | b) C++ 编译器  |
| b) ALU  | c) C++ 预处理器 |
| d) 输入单元 | e) 一个编辑器程序  |

1.11 为什么人们希望用一种与机器无关的语言来写程序,而不愿使用依赖于特定机器的语言?但在编写某些特殊类型的程序时,为什么采用依赖于机器的语言又要好一些?

1.12 填空题:

- 计算机的哪个逻辑单元负责从计算机外接收信息,以便由计算机使用? \_\_\_\_\_。
- 指示计算机解决特定问题的过程叫做\_\_\_\_\_。
- 哪种类型的计算机语言采用英语风格的缩写形式来表示机器语言指令? \_\_\_\_\_。
- 计算机的哪个逻辑单元负责将已由计算机处理好的信息发送给各种设备,以便那些信息能在计算机的外部使用? \_\_\_\_\_。
- 计算机的哪个逻辑单元负责保持信息? \_\_\_\_\_。
- 计算机的哪个逻辑单元负责执行计算? \_\_\_\_\_。
- 计算机的哪个逻辑单元负责做出逻辑决策? \_\_\_\_\_。
- 对程序员来说,能快速和方便地编写程序的一个计算机语言级别是\_\_\_\_\_。

- i) 计算机惟一能直接理解的语言叫做计算机的\_\_\_\_\_。
- j) 计算机的哪个逻辑单元负责协调其他所有逻辑单元的活动? \_\_\_\_\_。

1.13 指出下述每个对象的含义:

- a) `std::cin`
- b) `std::cout`
- c) `std::cerr`

1.14 今天,人们为什么普遍开始关注面向对象编程,特别是C++?

1.15 填空题:

- a) \_\_\_\_\_用于为程序编档,并改善其可读性。
- b) 用于在屏幕上打印信息的对象是\_\_\_\_\_。
- c) 用于做出决定的一个C++语句是\_\_\_\_\_。
- d) 计算通常由\_\_\_\_\_语句执行。
- e) 对象从键盘输入值。

1.16 针对下述任务,编写C++语句或代码行:

- a) 打印消息“Enter two numbers”。
- b) 将变量 `b` 和 `c` 的乘积赋给变量 `a`。
- c) 指出程序要执行一个示范性的工资计算(也就是说,用文字帮助编制程序文档)。
- d) 从键盘输入3个整数,并分别赋给整数变量 `a`、`b` 和 `c`。

1.17 判断正误。如果有错,请说明原因。

- a) C++ 操作符按从左到右的顺序求值的。
- b) 下面列出的都是有效的变量名:  
`_unser_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`。
- c) `cout << "a = 5;";` 该语句是赋值操作符的一个典型例子。
- d) 对于不含圆括号的一个有效C++算术表达式来说,它是按从左到右的顺序求值的。
- e) 下面列出的所有都是无效的变量名: `3g`, `87`, `67h2`, `h22`, `2h`。

1.18 填空题:

- a) 哪些算术操作符拥有与同乘法相同的优先级? \_\_\_\_\_。
- b) 圆括号嵌套时,算术表达式中哪些括号会先进行求值? \_\_\_\_\_。
- c) 对于计算机内存中的一个位置来说,假如它可能在程序执行的不同时刻包含不同的值,那么这个位置叫做一个\_\_\_\_\_。

1.19 执行下述每条C++语句时,会打印什么结果(如果可能的话)? 如果不打印任何内容,则答案是“不打印”。假定 `x = 2`, 而且 `y = 3`。

- a) `cout << x;`
- b) `cout << x + x;`
- c) `cout << "x = ";`
- d) `cout << "x = " << x;`
- e) `cout << x + y << " = " << y + x;`
- f) `z = x + y;`

- g) `cin >> x >> y;`
- h) `//cout << "x - y = " << x + y;`
- i) `cout << "\n";`

1.20 下面哪条C++ 语句中包含其值已被替换的变量?

- a) `cin >> b >> c >> d >> e >> f;`
- b) `p = i - j + k + 7;`
- c) `cout << "variables whose values are destroyed";`
- d) `cout << "a = b";`

1.21 假定一个代数方程式是  $y = ax^3 + 7$ 。下面哪条语句是其对应的C++ 表达式?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`
- e) `y = a * (x * x * x) + 7;`
- f) `y = a * x * (x * x + 7);`

1.22 指出下述每条C++ 语句中的操作符求值顺序,并在每条语句执行之后显示 x 的值。

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

1.23 写一个程序,要求用户输入两个数字,从用户处获得数字,并打印两个数字的和、乘积、差和商。

1.24 写一个程序,在同一行上打印 1~4 的数字,用一个空格分隔每一对相邻的数字。用下述方法写这个程序:

- a) 使用 1 个输出语句,和 1 个流插入操作符;
- b) 使用 1 个输出语句,和 4 个流插入操作符;
- c) 使用 4 个输出语句。

1.25 写一个程序,要求用户输入两个整数,从用户处获得数字,然后打印其中最大值,后面跟随“is larger.”字样。如输入的数字相等,则打印消息“These numbers are equal.”。

1.26 写一个程序,令其从键盘输入 3 个整数,并打印它们的和、平均值、乘积、最小值和最大值。屏幕对话如下所示:

```
Input three different integers:13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

1.27 写一个程序,用它读入一个圆的半径,并打印圆的直径、周长和面积。至于  $\pi$ , 请为其使用 3.141 59 这个常量值。在输出语句中执行这些计算(注意:在本章,我们只讨论了整数常量和变量。在第 3 章,我们还会讨论浮点数——亦即可以有小数点的值)。

1.28 写一个程序,用它分别打印一个矩形、一个椭圆、一个箭头以及一个菱形,如下所示:

```

*****
*       *
*       *
*       *
*       *
*       *
*       *
*       *
*****

      ***
    *   *
   *   *
  *   *
 *   *
*   *
*   *
*   *
  *   *
   *   *
    *   *
      ***

      *
     ***
    *****
     *
    *
   *
  *
 *
*
*
*
*
*

      *
     * *
    * *
   * *
  * *
 * *
* *
* *
* *
* *
* *
* *

```

1.29 指出下述代码的打印结果。

```
cout << " * \n * * \n * * * \n * * * * \n * * * * * \n";
```

- 1.30 写一个程序,读入 5 个整数,判断并打印其中的最大值和最小值。请只使用本章介绍的编程技术。
- 1.31 写一个程序,它能读入一个整数,判断并打印它是奇数还是偶数(提示:使用求模操作符。偶数必然是 2 的一个整数倍数。任何 2 的整数倍在被 2 除时,必然得到一个为 0 的余数)。
- 1.32 写一个程序,它能读入两个整数,判断并打印第一个数字是否为第二个数字的整数倍数(提示:使用求模操作符)。
- 1.33 用 8 个输出语句显示一个棋盘图案,然后用尽量少的输出语句,显示同一图案。

```

* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *

```

- 1.34 区分“致命错误”和“非致命错误”。说说为什么宁愿遇到致命错误,也不愿遇到非致命错误?
- 1.35 本题的内容可能有点儿超前。在本章,你学习了整数和 int 类型。C++ 还可以表示大写字母、小写字母以及数量可观的特殊符号。C++ 在内部使用小整数来表示每个不同的字符。一个计算机所支持的一系列字符,以及那些字符相应的整数表示形式,它们统称为那个计算机所采用的字符集。你为了打印一个字符,可把它简单地封闭在一对单引号中,如下所示

```
cout << 'A';
```

为了打印一个字符的整数表示,可使用 static\_cast,如下所示

```
cout << static_cast<int>('A');
```

这叫做强制类型转换运算(我们将从第 2 章正式介绍强制类型转换)。上述语句执行时,会打印出 65 这个值(在采用 ASCII 字符集的系统上)。请写一个程序,打印一些大写字母、小写字母、数位和特殊符号的整数形式。最起码,请打印出以下字符的整数形

式: A B C a b c 0 1 2 \$ \* + / 以及空字符。

- 1.36 写一个程序,它能输入一个数字,数字中包含5个数位。把那个数字分解成单独的数位,并打印出每一个数位,相互间用3个制表位分隔(提示:使用整除和求模操作符)例如,假定用户键入42 339这个数字,那么程序应打印结果

```
4 2 3 3 9
```

- 1.37 只用本章介绍的技术,写一个程序计算从0到10的数字的平方和立方,并利用制表位打印值表:

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

- 1.38 简要回答下述“对象思想”问题:

- 本书为什么要先详细讨论结构化编程,然后再深入探讨面向对象编程?
- 对一个面向对象的设计过程来说,一般包括哪些步骤?
- 人类如何表现多重继承?
- 人们相互间发送哪类的消息?
- 对象通过良好定义的接口相互间发送消息。那么,一部车载收音机(对象)向它的用户(即对象为“人”)展示了哪些接口?

- 1.39 你手上可能正戴着如今最流行的一类对象——手表。请阐述如何将以下的术语和概念应用于手表:对象、属性、行为、类、继承(例如,可设想一个闹钟)、抽象、建模、消息、封装、接口、信息隐藏、数据成员和成员函数。

## 第2章 控制结构

### 学习目标

- 理解基本的解决问题的方法
- 会利用“自上而下求精法”,开发自己的算法
- 会使用 if,if/else 和 switch 选择结构在备选的行动中选择
- 会用 while,do/while 和 for 等重复结构在一个程序中重复地执行语句
- 理解由计数器控制的重复,以及由标记控制的重复
- 会使用自增、自减、赋值和逻辑操作符
- 会使用 break 和 continue 程序控制语句

### 2.1 简介

为解决某个特定问题而专门编写程序之前,首先有必要全面理解这个问题,并周密计划每一步操作,力求最终圆满达到解决问题之目标。写程序时,相当重要的一点是理解基本构建单元的类型,并严格遵守业已证明行之有效的程序构建模型及规则。在本章,我们打算讨论这些问题,描述结构化编程的理论及原理。注意本章描述的技术适用于包括C++在内的大多数高级编程语言。从第6章开始学习C++的面向对象编程时,你会发现第2章学习的控制结构将非常有助于构建和操纵对象。

### 2.2 算法

针对任何计算问题,都可按某种顺序采取一系列行动来解决。解决问题的过程包含两方面的含义:

- (1) 要采取的行动;
- (2) 采取这些行动的顺序。

我们将这种过程称为算法。下面,让我们以实例来揭示正确指定行动顺序之重要性。

假定我们要设计一个“起床上班算法”,要求模拟一位公司职员早上起床并去上班的过程。他的行动如下:

- (1) 起床;
- (2) 脱下睡衣;
- (3) 洗个淋浴;
- (4) 穿好衣服;
- (5) 吃早餐;

(6) 开车上班。

只有按以上顺序采取各项行动,这位职员才能取得最高的效率。假如我们稍微调换一下顺序,变成:

- (1) 起床;
- (2) 脱下睡衣;
- (3) 穿好衣服;
- (4) 洗个淋浴;
- (5) 吃早餐;
- (6) 开车上班。

那么可怜的职员只好穿着湿淋淋的衣服去上班了!在计算机程序中,我们指定语句执行顺序的操作称为程序控制。本章将探讨C++的程序控制能力。

## 2.3 伪代码

伪代码(Pseudocode)是一种人工的、非正式的语言,目的是辅助程序员设计算法。利用我们在这里提供的伪代码,可有效地开发出实际、有效的程序算法,并可方便地转换成结构化的C++程序。伪代码像极了我们的口头语言,既方便,又好用——尽管并不是真正的计算机编程语言。

显然,用伪代码写的程序实际是无法在计算机上执行的。它们惟一的用处便是在用C++这样的程序语言写正式代码之前,帮助程序员事先“思考”并“把握”整个程序的执行流程。本章展示了几个例子,阐释了如何利用伪代码这一“利器”,高效率地开发出结构良好的C++程序。

在此,我们提供的伪代码程序纯粹由字符构成,所以程序员用一个编辑器程序即可方便地输入这些代码。计算机可极据需要显示一个伪代码程序的最新副本。对精心设计的伪代码程序来讲,它们可轻松转换成相应的C++程序。许多情况下,只需用对应的C++语句替换掉伪代码语句即可。

伪代码只由可执行语句构成——亦即一旦程序从伪代码转换成C++并开始运行,便会被执行的语句。声明并不是可执行语句。比如声明

```
int i;
```

其作用只是告诉编译器变量*i*的类型是什么,并指示编译器在内存中为此变量预留空间。但是,当程序执行时,这一声明并不会导致采取任何行动——比如输入、输出或者一次计算等等。有的程序员习惯于在伪代码程序的开头,列出要用的所有变量,并简要描述其用途。

## 2.4 控制结构

通常,程序中的语句是按原来书写的顺序,逐条执行的。我们把这种程序执行方式称为顺序执行。然而,我们即将讨论的许多C++语句还允许程序员自行控制接下去要执行的语句,该语句可能是、也可能不是顺序的“下一条”语句。我们把这样的程序执行方式称为转交控制权。



尽管这为编程带来了更大的灵活性,但在 20 世纪 60 年代,事实已证明胡乱转交控制权,也会加大软件开发的难度。`goto` 语句便是直接原因。利用这条语句,程序员可将控制权转交给一个程序里的几乎任意地方(跳到那个地方执行)。但由于它过度自由,所以也带来了混乱。因此,结构化编程的概念问世时,便郑重宣称自己是“免 `goto`”的。

Bohm 和 Jacopini 的研究证明<sup>①</sup>,程序完全可以不用任何 `goto` 语句。大势所趋下,程序员们也逐渐放弃对 `goto` 的“偏爱”,逐渐转向“无 `goto` 编程”。直到 20 世纪 70 年代,他们才开始认真看待“结构化编程”。研究结果令人振奋。采用了结构化编程后,程序员们普遍的反映是,一个软件项目无论开发时间、速度还是经济成本,都得到了有效的改善。之所以取得这些成功,是由于结构化程序条理更清晰,更易调试,更易修改,而且可有效地在第一时间避免出现大多数编程错误。

Bohm 和 Jacopini 的研究证明,所有程序其实只需 3 种控制结构。这 3 种控制结构包括:顺序结构、选择结构以及重复结构。其中,顺序结构是 C++ 内建的。除非特别声明,否则计算机都会按原先书写的顺序逐条执行 C++ 语句。图 2.1 的流程图为大家演示的便是一个典型的顺序结构,两个计算将依序执行。

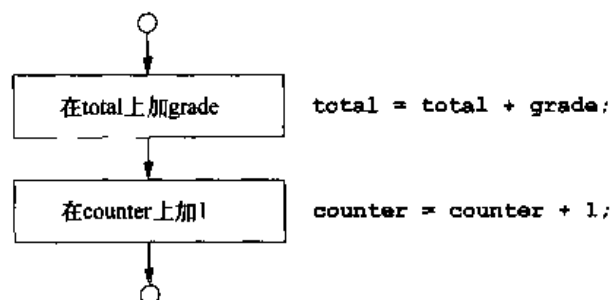


图 2.1 C++ 顺序结构流程图

在这里,流程图(Flowchart)是对整个程序算法或者一部分算法的图形化表示。在流程图中,我们往往会使用一些具有特殊意义的符号,比如矩形、菱形、椭圆以及小圆圈等等。在不同的符号之间,我们用一些箭头线进行连接,这些箭头线叫做流线(Flowline)。

与伪代码类似,流程图在开发和研究算法的时候特别有用。不过,当真正设计算法的时候,程序员最喜欢用的还是伪代码。流程图明确指出了控制结构的运作机制,这也是我们在本章要讨论的主题。

请观察图 2.1 的顺序结构流程图。我们用矩形符号(或称行动符号)指出每种类型的行动,包括计算或输入/输出操作等。图中的流线指出要采取的行动按什么顺序执行。首先,将 `grade` 加到 `total` 身上,再把 1 加到 `counter` 身上。在 C++ 顺序结构中,我们可采取的行动数量是无限的。稍后你会看到, C++ 程序里凡是需要采取一个行动的地方,都可按顺序采取几项行动。

如绘制流程图来表示一套完整算法,那么流程图的第一个符号应该是一个椭圆符号,在其中写上“开始”字样;最后一个符号也是一个椭圆符号,其中应写上“结束”字样。但如图 2.1

<sup>①</sup> Bohm, C. 和 G. Jacopini 的“Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”一文,见《Communications of the ACM》,第 9 卷,第 5 期,1996 年 5 月,336 ~ 371 页

所示,假如绘制的只是算法的一部分,那么头和尾的椭圆符号都可省略,分别换成一个小圆圈符号(或称接头符号)。

在所有流程图符号中,最重要的或许是菱形符号(或称决定符号),它描述了要在那个位置做出的一项决定。在下一节里,我们将更深入地讨论菱形符号。

C++ 提供了 3 种类型的选择结构。其中,假如符合某个条件(条件为 true),if 选择结构会执行(选择)一项行动;假如不符合(条件为 false),则跳过这项行动。在条件符合时(true),if/else 选择结构会采取一项行动;条件不符合时(false),则采取另一项行动。switch 选择结构则在许多不同的行动中选择一个,具体由一个整数表达式的值来决定。

if 选择结构是一种单选结构——它要么选择、要么忽略一项行动;if/else 是一种双选结构——它在不同的两项行动中做出选择;switch 则是一种多选结构——它在许多不同的行动中选择要执行的行动。

C++ 提供了 3 种类型的重复结构,分别是 while、do/while 和 for。要注意的是,对 if、else、switch、while、do 和 for 等单词来说,它们均是 C++ 的关键字。这些词是由语言为自己而保留的,用于实现像 C++ 控制结构这样的各种特性。千万不要将关键字用作标识符,比如变量名等等。图 2.2 展示了一个完整的 C++ 关键字列表。

C++ 关键字				
C 和 C++ 程序语言共用的关键字				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			
只限于 C++ 的关键字				
asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

图 2.2 C++ 关键字

### 常见编程错误 2.1 将关键字用作标识符是语法错误。

好了,这里可以简单总结一下。C++ 总共有 7 种控制结构,分别是:顺序结构、3 类的选择结构以及 3 类重复结构。每个 C++ 程序其实都是用这些控制结构搭建起来的。作为程序员,我们应根据程序需要实现的算法,挑选最恰当的控制结构。与图 2.1 展示的顺序结构一样,用一个流程图来表示控制结构时,每个控制结构都有两个小圆圈。一个圆圈在控制结构的入口处,另一个在出口处。利用这种单入/单出控制结构,我们可方便地构建程序。只需将一个控制结构的出口同下一个控制结构的入口连接在一起,便可将不同的控制结构有机

地联系到一起。事实上,类似的程序构建方法非常类似于小孩搭积木,所以我们将其简称为控制结构堆叠(Control - Structure Stacking)。不过,将不同的控制结构联系到一起时,这也并非惟一的方法。另外还有一种方法叫做控制结构嵌套(Control - Structure Nesting),详情参见后文描述。

**软件工程知识 2.1** 无论过去还是将来,我们的所有C++ 程序都可基于前述 7 类控制结构(顺序,if,if/else,switch,while,do/while 和 for)进行构建。不同控制结构只需通过控制结构堆叠和嵌套这两种方法即可合并到一起。

## 2.5 if 选择结构

利用选择结构,我们可在大量候选的行动中挑选一个。例如,假定通过一次考试的分数线是 60 分,那么伪代码应该如下所示:

假如学生成绩大于或等于 60

打印“通过”

在上述代码中,判断的是“学生成绩大于或等于 60”这个条件为 true(真)还是 false(假)。如条件为 true,则打印一条“通过”信息,然后按顺序“执行”下一条伪代码语句(记住伪代码并不是真正的程序语言)。如条件为 false,则忽略打印语句,并按顺序执行下一条伪代码语句。注意这个选择结构的第二行进行缩进处理。像这样的缩进并非必需的,但却是我们强烈建议的,因为它强调了结构化程序的固有的结构。将伪代码转换成真正的C++ 代码时,C++ 会忽略缩进以及空行中采用的任何空白字符(空格、制表符和换行符),所以空白字符的使用主动权完全掌握在程序员手中。善于利用它们,便可让自己的程序更富有条理、更易阅读。

**良好编程习惯 2.1** 在程序中合理进行缩进(缩排)处理,可显著增强程序的可读性——建议将每个缩进单位设为 1/4 英寸或 3 个空格字符。

在C++ 中,上述伪代码形式的 if 语句可写为

```
if ( grade >= 60 )  
    cout << "Passed";
```

注意,C++ 代码与伪代码严格对应。正是由于这种一一对应的关系,才使得伪代码成为强有力的一种开发工具。

**良好编程习惯 2.2** 在程序设计阶段,通常先用伪代码来“思考”一个程序,再将伪代码程序转换成真正的C++ 程序。

图 2.3 的流程图向大家揭示了单选 if 结构。在这个流程图中,包含了或许是最重要的流程图符号:菱形符号——或称决定符号,它表示在此需要做出一项决定。在判断符号中,包含了一个表达式(比如条件),它既可为 true,亦可为 false。从决定符号开始,必须引出两条流线:一个指出表达式为 true 时的程序执行方向,另一个指出表达式为 false 时的程序执行方向。通过第 1 章的学习,你知道决定可基于包含了关系或相等性操作符的条件。实际上,一项决定可基于任何表达式——如表达式的值为零,就把它当作 false;如表达式的值不是零(非零),就把它当作 true。C++ 标准专门提供了 bool 数据类型,用于代表 true 和 false。

true 和 false 这两个关键字用于表示 bool 值。

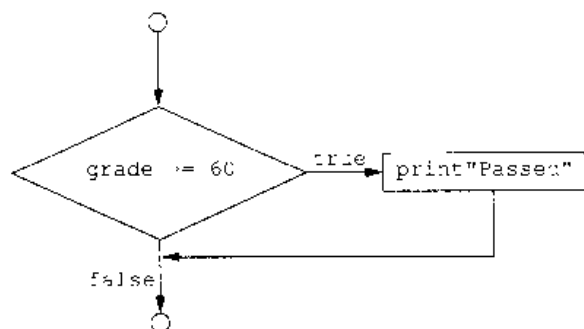


图 2.3 单选 if 结构的流程图

还要注意对于 if 结构来说,它是一种“单人/单出”控制结构。以后随着学习的深入,大家会知道对于各种控制结构来说,流程图中只能包含矩形和菱形符号(小圆圈和流线除外)——前者指出要采取的行动,后者指出要做出的决定。我们把这种系统称为行动/决定编程模型。

假设我们有 7 个柜子,每个都包含了 7 类控制结构中的一类。这些控制结构是空的。无论在矩形还是在菱形符号中,均不写任何内容。那么,作为程序员,我们的任务便是在惟一可选的两种方法中挑选一种(堆叠或嵌套),将这些控制结构合并到一起。然后,依据自己事先计划好的算法,用适当的方式来编制具体的行动及决定内容。稍后,我们便会全面讲解用来编制行动及决定内容的各种方式。

## 2.6 if/else 选择结构

只有条件为 true,if 选择结构才会执行指定的行动;否则便跳过此项行动。利用 if/else (假如/否则)选择结构,程序员可针对条件成立与不成立这两种情况,分别指定一项行动。伪代码语句

```

假如学生成绩大于或等于 60
    打印“通过”
否则
    打印“未通过”
  
```

中,学生成绩如高于或等于 60 分,则打印“通过”;如低于 60 分,则打印“未通过”。不管哪种情况,在完成打印操作后,都会依序执行下一条伪代码语句。注意在伪代码中,else(否则)的主体代码也进行了缩进处理。

**良好编程习惯 2.3** if/else 结构的两个主体语句都应缩进。

不管选择哪种缩进规范,都应在自己的所有程序中贯彻实施。阅读未能采用统一间距的程序是非常困难的。

**良好编程习惯 2.4** 如同时有数级缩进,那么每级缩进都在上一级缩进的基础上增加相同数量的空格。

换用C++, 上述伪代码可写成下面这样的正式 if/else 结构:

```
if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
```

参考图 2.4 的流程图, 大家可对 if/else 结构的控制流程一目了然。同时还请注意, 除了小圆圈和箭头(流线)之外, 流程图里只能用矩形(标明行动)和菱形(标明决定)。为加深大家的印象, 我们以后还会不断地强调这种行动/决定编程模型, 望大家不要感到厌烦。同样地, 大家可想象在一个柜子里, 包含了大量双选结构, 这些结构可能是构建一个C++ 程序所必需的。作为程序员, 我们的任务便是将这些双选结构同既定算法所需的其他控制结构合并到一起(采用堆叠或嵌套方式), 再用适用于目标算法的行动及决定内容来分别填充所有的空白矩形及菱形符号。

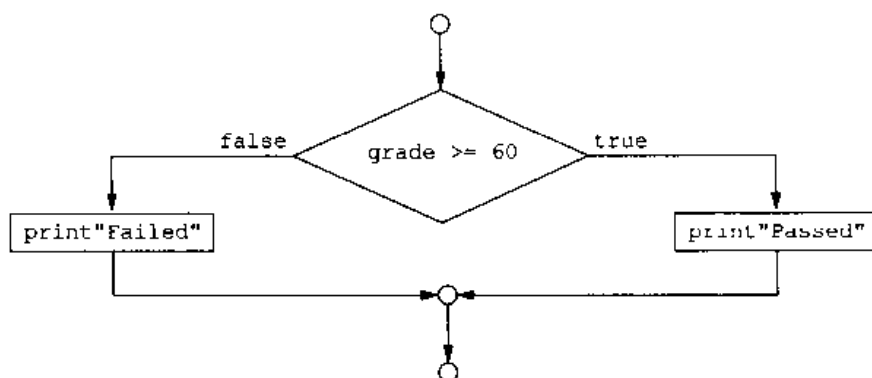


图 2.4 双选 if/else 结构的流程图

另外, C++ 还为我们提供了一个条件操作符(? :), 它大致相当于 if/else 结构。条件操作符是C++ 惟一的三元操作符, 它需要用到三个操作数。这些操作数同条件操作符组合到一起, 便构成了一个完整的条件表达式。第一个操作数乃一个条件; 第二个操作数指出条件为 true 时, 整个条件表达式的值; 第三个操作数指出条件为 false 时, 整个条件表达式的值。比如输出语句

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

包含了一个条件表达式——假如  $grade \geq 60$  这个条件成立, 表达式的值便是一个字符串“Passed”; 假如条件不成立, 表达式的值则为字符串“Failed”。换言之, 语句和条件操作符联合使用, 可实现与前述 if/else 语句相同的功能。不过正如大家马上就会看到的那样, 条件操作符的优先级比较低, 所以建议你无论如何都为条件加上一对圆括号, 就像上面那样(即使并非必需), 从而确保它得以优先执行。

条件表达式的值也可以是打算采取的行动(要执行的操作)。比如条件表达式

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

意思是: “假如 grade 大于或等于 60, 那么  $cout \ll "Passed"$ ; 否则的话,  $cout \ll "Failed"$ ”。这仍然类似于前述的 if/else 结构。我们马上便要看到, 条件操作符还可用在 if/else 语句无法胜任的一些地方。

嵌套 if/else 结构检测的是是否满足多个条件, 它将一个 if/else 选择结构放在另一个

if/else选择结构中。例如,下面的伪代码语句将为成绩大于或等于90分的打印 A;为范围在80~90分之间的打印 B;为范围在70~79分之间的打印 C;为范围在60~69分之间的打印 D;其他成绩则打印 F

```

假如学生成绩大于或等于 90
    打印 "A"
否则
    假如学生成绩大于或等于 80
        打印 "B"
    否则
        假如学生成绩大于或等于 70
            打印 "C"
        否则
            假如学生成绩大于或等于 60
                打印 "D"
            否则
                打印 "F"

```

上述伪代码用C++ 写成,便是

```

if ( grade >= 90 )
    cout << "A";
else
    if ( grade >= 80 )
        cout << "B";
    else
        if ( grade >= 70 )
            cout << "C";
        else
            if ( grade >= 60 )
                cout << "D";
            else
                cout << "F";

```

假如 grade 大于或等于 90,那么头四个条件都会是 true,但只有第一次检测后的 cout 语句才会执行。在那个 cout 执行之后,外层 if/else 语句的 else 部分就会被跳过不计。许多 C++ 程序员都喜欢将上述 if 结构写为

```

if ( grade >= 90 )
    cout << "A";
else if ( grade >= 80 )
    cout << "B";
else if ( grade >= 70 )
    cout << "C";
else if ( grade >= 60 )
    cout << "D";
else cout << "F";

```

这两种形式的效果是一样的。后者之所以较流行,是由于它避免了将代码过于向右缩进。过于向右缩进通常会在一行内留下较少的空间,导致强迫进行换行,从而降低程序的可读性。

**性能提示 2.1** 与罗列大量单选 if 结构相比,只用一个 if/else 结构的速度会快许多。这是由于在后一种情况下,只要碰到满足条件的第一个表达式,整个结构便会中止并退出,不必遍历所有表达式。

**性能提示 2.2** 在一个嵌套的 if/else 结构中,应首先检测最有可能为 true 的条件。这样一来,程序便可早早地退出 if/else 结构。与先检测不大容易成立的条件相比,这样的设计可显著加快执行速度。

if 选择结构认为自己主体内只有一条语句。要在一个 if 主体中包括几条语句,应将这些语句封闭在一对花括号中({和})。包含在一对花括号中的一系列语句统称为“复合语句”。

**软件工程知识 2.2** 程序内凡是放置单条语句的地方,都可放置复合语句。

下例中,我们便运用了复合语句,它们位于 if/else 结构的“else”部分:

```
if ( grade >= 60 ) {
    cout << "Passed.\n";
}
else {
    cout << "Failed\n";
    cout << "you must take this course again.\n";
}
```

在这种情况下,假如 grade 的值小于 60,程序就会连续执行 else 主体部分的两条语句,并打印出下面两条消息

```
Failed.
You must take this course again.
```

注意围绕于 else 从句中两条语句的花括号。这些花括号相当重要。如果没有花括号,语句

```
cout << "you must take this course again.\n";
```

就会转到 if 结构的 else 部分的主体外部,导致程序出现语法或逻辑错误。

**常见编程错误 2.2** 如忘记用一个或两个花括号为复合语句定界,会导致语法或逻辑错误。

**良好编程习惯 2.5** 始终记得在一个 if/else 结构(或其他任何控制结构)中放置花括号,这样有助于避免它们不慎被遗忘,特别是在以后为 if 或 else 从句添加语句的时候。

语法错误会被编译器捕捉。逻辑错误进入执行期才会表现出其效果。严重的逻辑错误会导致程序执行失败,并提前中止运行。而一个非严重逻辑错误允许程序继续执行,但可能产生不确切的结果。

**软件工程知识 2.3** 尽管前文提到“凡是放置单条语句的地方,都可放置复合语句”,但也不排斥另一种情况——根本不放置任何语句。也就是说,这些地方可以放置一条空语句。空语句的意思是指:在本该是语句的地方,单单用一个分号(;)来充数,而不采取任何行动。

**常见编程错误 2.3** 在 if 结构中,如在条件之后放置分号,那么假如是单选 if 结构,会造成逻辑错误;假如是双选 if 结构,则会造成语法错误(假如 if 部分包含了一个实际的主体语句)。

**良好编程习惯 2.6** 有的程序员在花括号内键入单独的语句之前,习惯于先输好复合语句的起始和结束花括号。这是一个好习惯,有助于避免不慎漏掉一个或两个花括号。

本节介绍了复合语句的记号法。在复合语句中可包含声明(就像 main 的主体一样)。如确实包含了声明,就称复合语句为一个块。块中的声明通常要摆在最前面,位于任何行动语句之前。但实际上,声明也可同行动语句混杂在一起使用。我们将在第 3 章讨论块的运用。不过在此之前,应尽量避免使用块(当然,main 的主体除外)。

## 2.7 while 重复结构

利用重复结构,程序员可指定一项行动。只要某些条件保持为 true,这个行动便会不断地执行下去。例如,可用伪代码进行以下描述:

只要(while)我的购物清单上还有货物

    购买下一种货物,并把它从清单上划掉

它表明购物时需要重复采取的操作。其中,“我的购物清单上还有货物”是一个条件,它可能成立(true),也可能不成立(false)。如果为 true,就采取“购买下一种货物,并把它从清单上划掉”这一行动。而且只要条件保持为 true,便重复地采取这一行动。while 重复结构中包含的语句便构成了 while 的主体。while 结构主体既可以是单独一条语句,也可以是一条复合语句。最终,我们设定的条件会变为 false(买了购物清单上的最后一件货物,并把它从清单上划掉)。在这个时候,重复会中止,并依序执行重复结构之后的第一条伪代码语句。

**常见编程错误 2.4** 在 while 结构的主体,假如不提供最终使 while 条件变成 false 的行动,会造成所谓的“无限循环”错误。在这种情况下,重复结构永远不会中止,会无休止地循环下去。

**常见编程错误 2.5** 如将关键字 while 误拼成 While,会导致语法错误(记住,C++ 是一种要严格区分大小写的语言)。对 C++ 的所有保留关键字而言,比如 while,if 和 else 等等,都只能采用小写字母。

为了让大家更深入地体会 while 循环的功用,在此不妨来考虑一个小程序,它的作用是计算 2 的所有乘方值,并判断哪个值首先大于 1 000。在此,假定整数变量 product 初始化为 2。while 重复结构

```
int product = 2;

while ( product <= 1000 )
    product = 2 * product;
```

执行完毕之后,product 变量便会包含预期的答案。

图 2.5 的流程图向大家揭示了与前述 while 结构对应的控制流程。同样地,请注意在流



程图中,只能包含矩形和菱形符号(小圆圈和箭头除外)。可设想在一个柜子中,包含了一系列空的 while 结构,它们可堆叠在其他控制结构上面,或与它们嵌套,从而以结构化的形式,实现某个具体算法的控制流程。随后,我们可用恰当的行动和决定来填充空矩形和菱形。这个流程图非常清晰地揭示了重复的机理。从矩形出发的流线会返回决定区域(菱形),除非某一个决定返回的结果为 false,否则每次都要对决定菱形中的条件进行检测。碰到 false 后,则马上退出 while 结构,并将控制权移交给程序中的下一条语句。

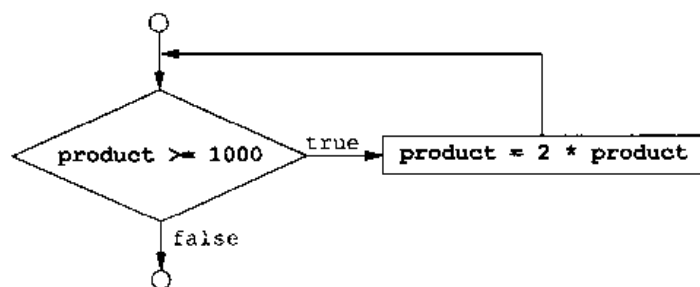


图 2.5 while 重复结构的流程图

初次进入 while 结构时,product 的值为 2。product 变量会不断地被 2 乘,所以被连续地赋予值 4,8,16,32,64,128,256,512 以及 1 024。一旦 product 的值变成 1 024, while 结构的“条件”(product <= 1 000)便不再成立,变成 false。因此,while 结构会中止并退出,因为 product 的终值为 1 024。程序将依序执行 while 之后的下一条语句。

## 2.8 算法设计:案例分析 1(计数器控制重复)

为阐释程序算法的制订过程,我们打算来研究一个求平均成绩问题的几种变化形式。先来看看下面这一条问题陈述:

有 10 名学生的一个班参加了一次测验。现在,成绩(0 到 100 之间的一个整数)已经给出。请计算平均成绩。

平均成绩等于成绩总和除以学生数量。要想用计算机解决这个问题,首先要制订一个算法。在这个算法中,要求先输入每个学生成绩,执行求平均值计算,然后打印出结果。

先用伪代码列出要在程序中采取的行动,同时指出以什么顺序来采取这些行动。在此,我们通过由计数器控制的重复来每次输入一个成绩。如采取这种重复方式,便需用到一个名为 counter(计数器)的变量,用它控制一系列语句重复执行的次数。在这个例子中,一旦计数器的值超过 10,便停止重复。在本节,我们用图 2.6 展示了伪代码算法,并用图 2.7 展示了具体的程序。在下一节,我们还准备讲解制订伪代码算法的具体过程。请注意,由计数器控制的重复通常也叫做确定重复(definite repetition)——这是由于在循环执行之前,我们便知道了准备重复的次数。

Set total to zero

Set grade counter to one

While grade counter is less than or equal to ten

Input the next grade

Add the grade into the total  
Add one to the grade counter

Set the class average to the total divided by ten  
Print the class average

图 2.6 通过计数器控制重复,解决平均成绩问题的伪代码算法

```

1 //Fig.2.7: fig02_07.cpp
2 //Class average program with counter - controlled repetition
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 |
11     int total,          //sum of grades
12     gradeCounter,      //number of grades entered
13     grade,              //one grade
14     average;           //average of grades
15
16     //initialization phase
17     total = 0;          //clear total
18     gradeCounter = 1;   //prepare to loop
19
20     //processing phase
21     while ( gradeCounter <= 10 ) {      //loop 10 times
22         cout << "Enter grade: ";      //prompt for input
23         cin >> grade;                  //input grade
24         total = total + grade;         //add grade to total
25         gradeCounter = gradeCounter + 1; //increment counter
26     }
27
28     //termination phase
29     average = total /10;                //integer division
30     cout << "Class average is " << average << endl;
31
32     return 0; //indicate program ended successfully
33 ;

```

输出结果:

```

Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82

```

```
Enter grade; 94
Class average is 81
```

图 2.7 通过计数器控制的重复来解决平均成绩问题时, C++ 程序代码及输出结果

在伪代码算法中, 注意我们引用了一个总成绩 (total) 和一个计数器 (counter)。其中, total 用于累加一系列值; 而 counter 是用来计数的一个变量——在这种情况下, 它用于统计输入了多少次成绩。用于存放 total 的变量在正式应用于程序之前, 通常应先初始化为零; 否则, 在总和里就会包括以前存放在 total 的内存位置处的值。

第 11 ~ 14 行

```
int total,                //sum of grades
    gradeCounter,         //number of grades entered
    grade,                //one grade
    average;              //average of grades
```

用于将 total、gradeCounter、grade 和 average 等变量声明为 int 类型。其中, 变量 grade 用于保存用户输入到程序中的值。

注意上述声明出现在 main 函数的主体中。记住, 在函数定义主体内声明的变量叫做局部变量, 它的使用范围 (作用域) 只能是从声明它的那一行起, 到函数定义的结束花括号 ({} ) 之前。要想在函数中使用一个局部变量, 必须先在那个函数中对其进行声明。

第 17 ~ 18 行

```
total = 0;                //clear total
gradeCounter = 1,         //prepare to loop
```

均是赋值语句, 用于将 total 初始化为 0, 将 gradeCounter 初始化为 1。

注意对 total 和 gradeCounter 这两个变量来说, 它们在一次计算中使用之前便得到了初始化。计数器往往初始化为 0 或 1——具体由其用途决定 (我们将用例子来说明每一种用途)。一个未初始化的变量包含着一个垃圾值 (也叫做未定义值)——内存位置上一次保存的值会随便地拿给那个变量使用。

**常见编程错误 2.6** 假如不初始化 counter 或 total, 会造成不正确的程序结果。这属于逻辑错误。

**良好编程习惯 2.7** 无论如何都要初始化计数器和总和。

**良好编程习惯 2.8** 单独用一行声明每个变量。

第 21 行

```
while ( gradeCounter <= 10 ) {           //loop 10 times
```

用于指出 while 结构应一直继续下去, 只要 gradeCounter 的值小于或等于 10。

第 22 行和第 23 行

```
cout << "Enter grade; ";                //prompt for input
cin >> grade;                            //input grade
```

它对应的是伪代码语句 “Input the next grade” (输入下一个成绩)。第一条语句在屏幕上显示提示信息: “Enter grade;” (输入成绩)。第二条语句则从用户那里输入成绩值。

接着, 程序通过用户输入的新的 grade 值来更新 total。第 24 行

```
total = total + grade;           //add grade to total
```

将 grade 加到 total 以前的值上,并将结果赋还给 total。

现在,程序已准备好对 gradeCounter 变量进行自增处理,指出已经处理完一条成绩,下面要从用户那里读取第二条成绩。所以第 25 行

```
gradeCounter = gradeCounter + 1; //increment counter
```

会将 1 加到 gradeCounter 上。这样一来,while 结构中的条件最终会变成 false,并中止循环。

第 29 行

```
average = total / 10;           //integer division
```

将求平均值的结果赋给变量 average。第 30 行

```
cout << "Class average is" << average << endl;
```

用于显示字符串“Class average is”,它的后面跟随变量 average 的值。

注意,程序中的求平均值计算最终产生了一个整数结果。实际上,这个例子中的成绩总和为 817。它被 10 除后,得到的结果应为 81.7,这是一个带小数点的数字。在下一节,你会看到如何处理这样的数字(即浮点数)。

**常见编程错误 2.7** 在一个由计数器控制的循环中,由于循环计数器(通过循环每次都累加 1 的时候)比它最后一个合法值大 1(在从 1 计数到 10 之后,变成 11),所以在循环之后的某次计算中,再使用计数器的值会造成一个“相差 1”错误。

在图 2.7 的程序中,假如第 29 行使用 gradeCounter 而不是用 10 来执行计算,程序的输出结果会变成不正确的 74。

## 2.9 算法设计:案例分析 2(标记控制重复)

下面,让我们稍微改动一下前面的求平均成绩问题,使其具有一般性:

开发一个求平均成绩程序,程序每次运行时,都可处理任意数量的成绩。

在第一个求平均成绩例子中,成绩的数量(10)是提前知道的。下面的例子则不事先假定要输入多少成绩。程序必须能处理任意数量的成绩。那么,程序何时应该停止成绩输入呢?什么时候应该计算,什么时候则应打印平均成绩呢?

为解决这些问题,一个办法是使用名为“哨兵”的一个特殊值——亦称作“信号值”、“假值”或者“标记值”等等(在英语中,分别叫作 Sentinel value、Signal value、Dummy value 或者 Flag value 等等。——译者注)。作为用户,需要不断输入成绩,直到所有合法成绩都输入完毕。随后,用户还得输入一个特殊的标记,向程序指出:以后不再有更多的输入啦!由“哨兵”控制的重复通常叫作不确定重复(indefinite repetition),因为在循环开始之前,根本不知道将要重复多少遍。

显然,挑选标记时,必须遵循特定的规则。否则,可能会与那些合法的输入值冲突。由于测验成绩通常是正整数,所以 -1 是可接受的标记。因此,运行一次该求平均成绩的程序,可以会出现类似“95,96,75,74,89 和 -1”的一连串输入。碰到 -1 后,程序应能马上计算并打印出 95,96,75,74 和 89 这 5 个值的平均值。换言之,由于 -1 是一个标记,所以不会参与求平均值的计算。

**常见编程错误 2.8** 如挑选的标记同时也是合法的输入数据,就会造成逻辑错误。

为进一步优化计算平均成绩的这个程序,我们采用了一种名为“自上而下求精法”的技术,即“Top-down, Stepwise Refinement”。利用这种技术,可以非常轻松地开发出良好结构化的程序。首先,让我们自上开始写出一条伪代码语句

判断测验平均成绩

在上部,是一条简单的语句,指出该程序的总体功用是什么。所以事实上,最上部的语句相当于对程序的一种“完整”表述。然而不幸的是,仅仅通过顶部这条消息(如上所示),实际看不出足够多的细节,只根据它是没法子写出一个完整C++程序的。为此,我们接下来要进行“求精”处理。首先,将顶部的语句分割为一系列更小的任务,然后按它们执行时的顺序排好。由此便得到了下面的第一部分“求精”结果

初始化变量

输入、总计和计数测验成绩

计算并打印平均成绩

这里只采用了顺序结构——上面列出的所有步骤会逐步执行。

**软件工程知识 2.4** 每次求精(包括最顶部的整体表述)其实都是一个完整的算法规范;只是细化程度有所不同。

**软件工程知识 2.5** 许多程序都可在逻辑上分成3个阶段:初始化阶段对变量进行初始化;处理阶段输入数据值,并相应地更改程序变量;结束阶段则计算并打印出最终结果。

其实就第一次求精来说,参照上述“软件工程知识”的提示进行操作便足够了。不过为了进行下一个级别的求精(二次求精),我们便需要用到一些特定的变量。在此,我们需要计算出数字的总和,统计总共处理了多少个数字,还要用一个变量来保存用户输入的每一个成绩,最后再用一个变量保存计算过后的平均值。所以对伪代码语句

初始化变量

进行“求精”后,可获得结果

将总和初始化为零

将计数器初始化为零

要注意的是,只有变量 total 和 counter 才需在使用前初始化;相反,average 和 grade 这两个变量(分别保存计算好的平均值和用户输入值)则毋需初始化——因为它们的值会在计算或输入之后,得到自动改写和更新。

另外,伪代码语句

输入、总计和计数测验成绩

要求用一个重复(即“循环”)结构来实现,以使用户连续输入每一个成绩。由于事先不知道总共要处理多少个成绩,所以我们计划用一个“由哨兵控制的重复”结构。使用这种结构,用户需要一次输入一个合法的成绩值。输入了最后一个合法的成绩后,需要在下一次重复的时候,输入标记。程序会在每次输入了成绩后检测其是否为标记;一旦判定为标记,循环就会中止。综上所述,上面那条伪代码语句的二次求精结果如下

输入第一个成绩(可能是标记)

只要(while)用户还没有输入标记

```

把这个成绩加到总和里
成绩计数器自增 1
输入下一个成绩(可能是标记)

```

注意,在伪代码中,我们不用花括号将构成 while 结构主体的语句封闭起来。在这里,只是简单地对 while 之下的语句进行缩进处理,以显示出它们从属于 while。再次提醒大家,伪代码只是一种辅助性的程序开发工具,主要作用是帮助你“思考”。

最后一条需要“二次求精”的伪代码语句

```

计算并打印平均成绩

```

其求精结果

```

假如(if)计数器不等于 0
    将平均值设为总和除以计数器值
    打印平均值
否则(else)
    打印“没有成绩被输入”

```

注意,我们在这里非常仔细地进行了检测,以避免出现“被零除”的情况。“被零除”是严重的逻辑错误。如未检测到这一错误,程序执行便会失败(程序“崩溃”)。图 2.8 展示了针对求平均成绩问题的伪代码进行二次求精的完整过程。

```

将总和初始化为零
将计数器初始化为零

输入第一个成绩(可能是标记)
只要(While)用户还没有输入标记
    把这个成绩加到总和里
    成绩计数器自增 1
    输入下一个成绩(可能是标记)

假如(if)计数器不等于 0
    将平均值设为总和除以计数器值
    打印平均值
否则(else)
    打印“没有成绩被输入”

```

图 2.8 通过标记控制的重复,解决求平均成绩问题的伪代码算法

**常见编程错误 2.9** 被零除会造成严重错误。

**良好编程习惯 2.9** 进行除法运算时,假如除数可能为零,就务必明确检测这一条件,并在程序中提前采取防范措施(比如打印一条出错提示信息等),不要让潜在的问题引发严重错误!

图 2.6 和图 2.8 中,我们在伪代码中包含了一些完整的空行,使伪代码便于阅读。空行可将程序分为几个明显不同的阶段。

图 2.8 的伪代码算法用于解决更为常规求平均成绩问题。这个算法是在只经过了两次“求精”之后开发出来的。有些情况下,可能还需要进一步的求精。

**软件工程知识 2.6** 只要伪代码算法提供了足够多的细节,利用这些细节可将伪代码轻松转

换成C++程序,便可考虑停止“自上而下求精”了。随后即可根据伪代码轻松编写C++程序。

图2.9展示了C++程序和一次演示执行结果。尽管只输入了整数成绩,但平均值计算有可能产生一个小数,即一个实数。`int`类型不能表示实数。所以程序采用了数据类型`double`来处理带有小数点的数字(即浮点数),并采用了一个特殊的操作符,名为强制类型转换操作符,强迫平均值计算产生一个浮点数结果。这些特性将在展示了程序之后详细说明。

```

1 //Fig.2.9: fig02_09.cpp
2 //Class average program with sentinel - controlled repetition.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::ios;
9
10 #include <iomanip>
11
12 using std::setprecision;
13 using std::setiosflags;
14
15 int main()
16 {
17     int total,           //sum of grades
18         gradeCounter,   //number of grades entered
19         grade;          //one grade
20     double average;      //number with decimal point for average
21
22     //initialization phase
23     total = 0;
24     gradeCounter = 0;
25
26     //processing phase
27     cout << "Enter grade, -1 to end: ";
28     cin >> grade;
29
30     while ( grade != -1 ) {
31         total = total + grade;
32         gradeCounter = gradeCounter + 1;
33         cout << "Enter grade, -1 to end: ";
34         cin >> grade;
35     }
36
37     //termination phase
38     if ( gradeCounter != 0 ) {
39         average = static_cast< double >( total ) / gradeCounter;
40         cout << "Class average is " << setprecision( 2 )
41             << setiosflags( ios::fixed | ios::showpoint )
42             << average << endl;
43     }
44     else

```

```

45     cout << "No grades were entered" << endl;
46
47     return 0; //indicate program ended successfully
48 ;

```

输出结果:

```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50

```

图 2.9 采用标记控制重复,求平均成绩问题的C++ 程序以及示范输出结果

在这个例子中,我们看到控制结构可以一个接一个按顺序堆叠起来,就像小孩搭积木一样。在程序中,while 结构(第 30 到 35 行)的后面紧跟着一个 if/else 结构(第 38 到 45 行)。该程序的许多代码与图 2.7 的代码完全相同,所以在这个例子中,我们只把重点放在新出现的一些特性和问题上。

第 20 行声明了 double 变量 average。通过这一改变,我们可将求平均成绩的结果保存为浮点数。第 24 行将变量 gradeCounter 初始化为 0,因为目前尚未输入任何成绩。记住,这个程序使用的是一个由标记控制的重复。为准确记录输入的成绩数量,只有在输入了一个有效的成绩值之后,gradeCounter 变量才会自增。

注意这两条一样的输入语句(第 28 行和第 34 行)

```
cin >> grade;
```

之前都有一条输出语句,提醒用户输入。

**良好编程习惯 2.10** 每次需要键盘输入时都提醒用户。提醒时,应指出输入所采取的形式,以及任何特殊值(比如用户应输入哪个标记中止循环)。

**良好编程习惯 2.11** 标记控制的循环中,须在要求输入数据的提醒中,明确指定标记。

研究一下由标记控制的重复,再对比一下图 2.7 由计数器控制的重复,我们可发现两种重复结构在程序逻辑上的差异。在由计数器控制的重复中,我们每遍历一次 while 结构(遍历的总次数是事先给定的),都要从用户那里读回一个值。而在由哨兵控制的重复中,在程序抵达 while 结构之前,我们都会读回一个值(第 28 行)。这个值用于判断程序的控制流程是否应该进入 while 结构的主体。假如 while 结构的条件为 false(表明用户已经键入了标记),那么 while 结构的主体便不再执行(表明未输入成绩)。但与此相反的是,假如条件为 true,就开始执行主体,而且开始处理用户输入的数值(本例是加入 total)。这个值处理完毕后,在抵达 while 结构主体的末尾之前,要求用户输入下一个值。在第 35 行,抵达了标志着主体部分结束的右花括号(}),就会开始对 while 结构的条件进行下一次测试。此时,会利用用户刚才输入的新值,判断 while 结构的主体是否应该重新执行。注意,下一个值肯定是



在检测 while 结构条件之前由用户输入的。这样一来,对于用户刚才输入的值来说,在正式对其进行处理之前(累加到 total 之前),就可事先判断它是不是标记。如这个值确属标记,则立即中止 while 结构,新输入的标记不会被错加到 total 上。

在图 2.9 中,要注意 while 循环中的复合语句。没有花括号,循环主体中的最后 3 条语句就会被排除在循环之外,导致计算机无法正确解释以下代码

```
while ( grade != -1 ) {
    total = total + grade;
    gradeCounter = gradeCounter + 1;
    cout << "Enter grade, -1 to end: ";
    cin >> grade;
```

假如用户不为第一个成绩输入 -1,就会造成一个无限循环。

求平均值的结果并非肯定为整数值。通常,平均值是一个包含了小数部分的值,比如 7.2 或 -93.5。这些值叫做浮点数,在 C++ 中用数据类型 float 和 double 表示。同类型为 float 的变量相比,类型为 double 的变量可保存的值较大且较为精确。考虑到这个原因,我们在程序中更喜欢使用 double,而不是 float。在 C++ 中,常量(比如 1 000.0 和 .05)被视为 double 类型。

变量 average 被声明为类型 double,以捕捉计算后的小数结果。然而,total / counter 这一计算的结果是一个整数,因为 total 和 counter 都是整数变量。两个整数相除便是所谓的整数除法。在这样的计算中,计算结果的小数部分会自动丢失(亦即被截掉)。由于计算是首先执行的,所以在结果分配给 average 之前,小数部分就已经丢失了。为了针对整数值而进行一次浮点计算,必须自行创建临时值,它们是计算时使用的浮点数。C++ 为此提供了一元强制类型转换操作符。语句

```
average = static_cast < double > ( total ) / gradeCounter;
```

包含了强制类型转换操作符 static\_cast < double > ( ),它可为括号中的操作数(total)创建一个临时性的浮点数副本。以这种方式使用一个强制类型转换操作符,叫做显式转换。存储在 total 中的值仍是一个整数。只是,现在的计算过程是一个浮点值(total 的临时 double 版本)被整数 counter 除。

C++ 编译器只知道如何计算操作数的数据类型完全一致的表达式。为保证操作数具有相同的类型,编译器会针对所选的操作数执行所谓的提升操作(也叫做隐式转换)。例如,在包含了数据类型 int 和 double 的一个表达式中,int 操作数被提升为 double。在我们的例子中,当 counter 被提升为 double 类型后,会开始执行计算,浮点除法的结果被赋给 average。本章稍后还会讨论所有标准数据类型及其提升顺序。

强制类型转换操作符适用于任何数据类型。其中,static\_cast 操作符的结构是:在 static\_cast 这个关键字后而跟上一对尖括号(< >),在括号中封闭上一个数据类型的名称。注意强制类型转换操作符是一种一元操作符;换言之,它是只需要取得一个操作数的操作符。在第 1 章,我们已经学习了二元算术操作符。C++ 还支持加号(+)和减号(-)操作符的一元版本,所以程序员可以编写像 -7 或 +5 这样的表达式。强制类型转换操作符具有比其他一元操作符更高的运算优先级,比如一元 + 和一元 - 等等。它的优先级高于乘法操作符 \*、/ 和 %,但要比括号的低。在我们的优先顺序表中,用 static\_cast < 类型 > ( ) 来表示强制类型

转换操作符。

图 2.9 的格式化功能将在第 11 章深入讲解,这里只作简要讨论。输出语句中的 `setprecision(2)` 调用

```
cout << "Class average is" << setprecision(2)
    << setiosflags( ios::fixed , ios::showpoint )
    << average << endl;
```

指出 `double` 变量 `average` 打印时将在小数点右侧保留 2 个数位的精度(比如 92.37)。我们将这样的调用称为参数化流操纵元。采用这种调用的程序必须包含一条预编译指令:

```
#include <iomanip>
```

第 11 行和第 12 行指出来自 `<iomanip>` 头文件的名称将用于程序中。注意 `endl` 是一个非参数化流操纵元,它不需要 `<iomanip>` 头文件。如果未指定精度,浮点值通常会采用 6 个数位的精度来输出(此为默认精度)。不过,稍后你可看到例外情况。

在上述语句中,流操纵元 `setiosflags( ios::fixed | ios::showpoint )` 设置了两个输出格式化选项,即 `ios::fixed` 和 `ios::showpoint`。竖线字符(`|`)用于对一个 `setiosflags` 调用中的多个选项进行分隔(第 16 章将深入讲解 `|` 记号的用法)。注意,尽管逗号(`,`)多用于分隔一系列项目,不可用于流操纵元 `setiosflags` 中;如果你使用逗号,那么只有列表中的最后一个选项才会被设置。`ios::fixed` 这个选项导致一个浮点值采用所谓的定点格式进行输出(与此相反的是科学记号法,详情参见第 11 章)。`ios::showpoint` 选项强制小数点和尾随的零都打印出来——即使是一个像 88.00 这样的整数。如果不设 `ios::showpoint` 选项,类似的值在 C++ 中会打印成 88,没有追尾的零和小数点。如果在一个程序中使用前述格式,打印出来的值会自动舍入处理,转换成要求的小数位数——但保留在内存中的值仍然不变。例如,87.945 和 67.543 将分别输出为 87.95 和 67.54。

**常见编程错误 2.10** 使用浮点数时,不可假定它们肯定能精确地表示,否则会导致不确切的结果。在大多数计算机上,浮点数都是约数。

**良好编程习惯 2.12** 不要试图比较两个浮点数是否相等。而应该测试两个浮点数的差值的绝对值是否小于一个指定的小值。

尽管浮点数并非肯定“百分百精确”,但它们仍有许多用武之地。举个例子来说,当我们说人体正常温度是 98.6 华氏度时,不需要精确到较多的数位。用一支温度计查看温度时,虽然读数是 98.6,但它实际可能是 98.599 947 321 064 3。将这个数字读作 98.6,已经可以满足大多数应用了。

浮点数是约数的特点还可通过除法表现出来。将 10 除以 3 时,结果是 3.333 333 3...。一系列 3 会无限重复下去。但是,计算机只分配了一个固定大小的空间来保存这样的值,所以很明显,浮点值只能表现出一个大概。

## 2.10 算法设计:案例分析 3(嵌套控制结构)

接下来分析另一个完整的例子。同样,我们先用伪代码技术和“自上而下求精法”设计一个算法,最后再根据它写出相应的 C++ 程序。通过以前的案例分析,我们知道控制结构既

可逐个按顺序堆叠起来(当然要),就像小孩搭积木那样。在这个案例分析中,我们则打算讲解在C++中,控制结构如何用另一种方法相互连接起来——将一个控制结构嵌套到另一个控制结构里。

下面是问题陈述:

一所大学提供了一门课,让学生准备本州的房地产中介资格考试。去年,完成了这门课的几名学生参加了资格考试。很自然,学校想知道自己的学生在考试时的表现。现在,要求你写一个程序,对考试结果进行总结。你得到10名学生的一个列表。在每个姓名旁边,1表示该学生通过了考试;2表示没有通过。

程序应该这样对考试结果进行分析:

(1) 输入每一个考试结果(1或2)。每次请求另一个考试结果时,都在屏幕上显示消息:“Enter result”(输入结果)。

(2) 统计两类考试结果的数量(1的数量和2的数量)。

(3) 显示考试结果总结,分别指出通过和没有通过考试的学生数量。

(4) 假如有8名以上的学生通过考试,便打印一条消息“Raise tuition”(提高学费)。

仔细地阅读上述问题陈述,我们可得出下述结论:

(1) 程序必须能处理10个考试结果,所以需要用一个由计数器控制的循环。

(2) 每个考试结果都是一个数字,要么是1,要么是2。程序每次读入一个考试结果时,都必须检测该数字是1还是2。在我们拟定的算法中,打算检测的是1。假如数字不是1,则假定它是2(在本章末的一个练习题中,我们将检讨这一假定所带来的后果)。

(3) 要使用两个计数器:一个统计通过考试的学生数量,另一个统计没有通过考试的学生数量。

(4) 程序处理完所有结果后,必须判断总共是否有8名以上的学生通过了考试。

下面将切入正题!我们要用功能强大的“自上而下求精法”构建具体的算法。首先当然是最“上”面的伪代码语句

分析考试结果,判断是否该提高学费

同样地,我们需要强调的是,最上面的这条语句是对整个程序的一个完整表述。不过请记住,为获得足够多的细节,以便轻松转换成C++程序,必须根据情况对其进行多次“求精”。第一次“求精”的结果

初始化变量

连续输入10个考试成绩,统计通过和没有通过考试的数量

打印考试结果总结,判断是否该提高学费

显然,以上的信息还不够“细”。虽然足以完整表述一个程序,但仍须进行更进一步的求精。首先来考虑一系列特定的变量。计数器显然是需要的,我们得用它们记录通过和未通过的数量。另外还要用一个计数器来控制循环,并用一个变量保存用户输入。伪代码

初始化变量

对它求精的结果如下

把 passes(通过考试)初始化为0

把 failures(未通过考试)初始化为0

把 studentCounter(学生计数器)初始化为1

请注意上面只初始化了计数器和总数。接下来是需要二次求精的语句

连续输入 10 个考试成绩,统计通过和没有通过考试的数量

这要求一个循环来连续输入每一个考试成绩。由于提前准确地知道所需的 10 个考试成绩,所以理所当然地应该使用计数器控制的循环(重复)结构。在循环内部(它嵌套于另一个循环内),我们用双选结构来判断每个考试结果到底是通过,还是没有通过。然后,有选择地令不同的计数器自增 1。二次求精后,得到伪代码语句

假如(While)学生计数器小于或等于 10  
输入下一个考试结果

假如(If)学生通过考试  
在 passes 上加 1  
否则(Else)  
在 failures 上加 1

在学生计数器上加 1

注意用空行将 if/else 控制结构独立出来,以改善程序的可读性。最后,我们需要对语句

打印考试结果总结,判断是否该提高学费

进行二次求精。求精结果为

打印通过考试的人数(passes)  
打印没有通过考试的人数(failures)

假如 8 名以上的学生通过  
打印"Raise tuition"(提高学费)

图 2.10 列举了完整的二次求精结果。注意空行也用于将 while 结构独立出来,以改善程序的可读性。

把 passes(通过考试)初始化为 0  
把 failures(未通过考试)初始化为 0  
把 studentCounter(学生计数器)初始化为 1

假如(While)学生计数器小于或等于 10  
输入下一个考试结果

假如(If)学生通过考试  
在 passes 上加 1  
否则(Else)  
在 failures 上加 1

在学生计数器上加 1

打印通过考试的人数(passes)  
打印没有通过考试的人数(failures)

假如 8 名以上的学生通过  
打印"Raise tuition"(提高学费)

图 2.10 考试结果问题的伪代码

从以上伪代码可以看出,由于已提供的细节非常丰富,所以完全可将其直接转换为一个真正的 C++ 程序。图 2.11 展示了 C++ 代码清单以及两次运行情况。

```
1 //Fig.2.11: fig02_11.cpp
2 //Analysis of examination results
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     //initialize variables in declarations
12     int passes = 0,      //number of passes
13         failures = 0,    //number of failures
14         studentCounter = 1, //student counter
15         result;          //one exam result
16
17     //process 10 students; counter - controlled loop
18     while ( studentCounter <= 10 ) {
19         cout << "Enter result (1 =pass,2 =fail): ";
20         cin >> result;
21
22         if ( result == 1 ) //if/else nested in while
23             passes = passes + 1;
24         else
25             failures = failures + 1;
26
27         studentCounter = studentCounter + 1;
28     }
29
30     //termination phase
31     cout << "Passed " << passes << endl;
32     cout << "Failed " << failures << endl;
33
34     if ( passes > 8 )
35         cout << "Raise tuition" << endl;
36
37     return 0; //successful termination
38 }
```

输出结果:

```
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):2
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
Enter result (1 =pass, 2 =fail):1
```

```

Passed 9
Failed 1
Raise tuition
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):2
Enter result (1 = pass, 2 = fail):2
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):2
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):1
Enter result (1 = pass, 2 = fail):2
Passed 6
Failed 4

```

图 2.11 考试结果问题的C++ 程序及输出结果

第 12 ~ 15 行

```

int passes = 0,           //number of passes

failures = 0,            //number of failures

studentCounter = 1,      //student counter

result;                  //one exam result

```

用于声明 main 中使用的变量,以便对考试结果进行处理。注意我们在此利用了C++ 的一个特点:允许变量初始化工作在声明时完成(passes 赋值为 1, failures 赋值为 0, 而 studentCounter 赋值为 1)。在每一次重复之前,可能都要求对循环程序进行初始化;像这样的初始化工作通常是在赋值语句中进行的。

**良好编程习惯 2.13** 声明时便对变量进行初始化,有助于程序员避免以后忘记对数据进行初始化的问题。

**软件工程知识 2.7** 经验表明,用计算机解决一个现实世界的问题时,最困难的部分便是设计出一个合理的算法。一旦拟出正确的算法,再把它转换成实际的C++ 程序,就非常简单!

**软件工程知识 2.8** 许多高手在写程序时,根本没用过像“伪代码”这样的辅助开发工具。这些程序员认为自己的终极目标便是用计算机解决实际问题。如果写伪代码的话,会耽搁时间,影响开发进度。不过要提醒大家的是:一方面,其实你可能并非那种意义上的“高手”;另一方面,这不用伪代码也确实可以解决简单而熟悉的问题,但对一些大型的、复杂的项目来说,恐怕会碰到很多难题。

## 2.11 赋值操作符

C++ 提供了几个赋值操作符,可用于对赋值表达式进行简写。语句

```
c = c + 3;
```

可用“+=”加后赋值操作符简写为

```
c += 3;
```

其中,操作符“+=”将右侧表达式的值加到左边的变量值身上,再把结果赋给左边的变量。总之,凡是碰到下面的类似语句

```
变量 = 变量 操作符 表达式;
```

(其中的“操作符”可为+、-、\*、/或%等任意一元操作符,或为我们稍后会讲到的其他某个操作符)其实都可简写为

```
变量 操作符 = 表达式;
```

因此,“c += 3”的意思应该是“把3加到c”。图2.12总结了一些典型的算术赋值操作符,列举并解释了一些简单的示例。

操作符	示例表达式	解释	赋值
假设: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 到 c
-=	d -= 4	d = d - 4	1 到 d
*=	e *= 5	e = e * 5	20 到 e
/=	f /= 3	f = f / 3	2 到 f
%=	g %= 9	g = g % 9	3 到 g

图 2.12 算术赋值操作符

**性能提示 2.3** 如使用了“简写”的赋值操作符,程序员就可以更快地编写程序,而编译器能更快地编译程序。在使用了“简写”赋值操作符的前提下,有的编译器可生成最终可较快运行得更快的代码。

**性能提示 2.4** 本书的许多“性能提示”只能稍微提升性能,所以有的读者可能不会把它们放在心上。但是,在一个需要重复多次的循环中,假如能每一次都“稍微”提升性能,最终带来的性能提升仍然可观。

## 2.12 自增和自减操作符

C++ 还提供了一元自增操作符“++”,以及一元自减操作符“--”,图2.13总结了它们的用法。假如变量c的值需要自增1,便可直接用自增操作符“++”来实现,无须写成“c = c + 1”或者“c += 1”。若将自增或自减操作符放在变量名前,则分别叫作前自增或前自减操作符。相反,如果将自增或自减操作符放在变量名后面,则分别叫作后自增或后自减操作符。如果对一个变量进行前自增/前自减处理,可促使它自动增/减1;随后,在变量所在的那个表达式内,开始启用它的新值。相反,如果对一个变量进行后自增/后自减处理,那么在变量所在的那个表达式内,将沿用它的旧值(当前值)——用了之后,再让它自动增/减1。

图2.14中的程序向大家展示了操作符“++”的前自增及后自增这两个版本间的差异。如对变量c实行后自增处理,就只有在输出语句中用过之后,它才会增值。相反,如对变量c实行前自增处理,就会先增值,再将新值应用于输出语句。

操作符	名称	示例表达式	说明
++	前自增	++a	预先在 a 的基础上加 1, 然后在 a 所在的表达式中使用 a 的新值
++	后自增	a++	在 a 所在的表达式中使用 a 的当前值后, 让 a 加 1
--	前自减	--b	预先在 b 的基础上减去 1, 然后在 b 所在的表达式中使用 b 的新值
--	后自减	b--	在 b 所在的表达式中使用 b 的当前值后, 让 b 减 1

图 2.13 自增和自减操作符

```

1 //Fig. 2.14: figC2_14.cpp
2 //Preincrementing and postincrementing
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int c;
11
12     c = 5;
13     cout << c << endl;           //print 5
14     cout << C++ << endl;         //print 5 then postincrement
15     cout << c << endl << endl;   //print 6
16
17     c = 5;
18     cout << c << endl;           //print 5
19     cout << ++c << endl;         //preincrement then print 6
20     cout << c << endl;           //print 6
21
22     return 0;                   //successful termination
23 }

```

输出结果:

```

5
5
6

5
5
6

```

图 2.14 前自增和后自增的差异

程序会在使用操作符“++”之前与之后, 分别显示 c 值。自减操作符“--”的作用与此类似。

**良好编程习惯 2.14** 一元操作符应紧挨操作数, 中间不含任何空格。

图 2.11 中的下述 3 个赋值语句

```

passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter - 1;

```



可利用赋值操作符简写为

```
passes += 1;
failures += 1;
studentCounter += 1;
```

或用前自增操作符简写为

```
++passes;
++failures;
++studentCounter;
```

或用后自增操作符简写为

```
passes++;
failures++;
studentCounter++;
```

要注意,在一条语句中自增或自减一个变量时,前自增和后自增的结果是相同的;而且前自减和后自减结果也相同。只有在一个较大的表达式中使用一个变量时,前自增和后自增变量才会出现不同的结果(前自减和后自减的情况类似)。此外,执行前自增和前自减时,速度会稍快于后自增和后自减。

目前,大家只须记住只有一个简单的变量名才可作为一个自增或自减操作符的操作数使用。我们不久会看到在所谓的“左值”(lvalues)中,也可使用这些操作符。

**常见编程错误 2.11** 若对比较复杂的变量名应用自增或自减操作符 C 如 `++(x+1)`,会出现语法错误。

图 2.15 展示了本书已讨论过的所有操作符的优先级以及结合性。其中,按从上到下的顺序,优先级依次降低。第二列指出在优先级相同的情况下,将按什么顺序(方向)执行不同的运算。请注意条件操作符(`?:`)、一元自增操作符(`++`)、一元自减操作符(`--`)、加(`+`)、减(`-`)和强制类型转换以及赋值操作符(`=`, `+=`, `-=`, `*=`, `/=` 和 `%=`)会按从右到左的方向结合。在图 2.15 中,其他所有操作符均会按从左到右的顺序。第 3 列指出了各操作符组别的名称。

操作符	结合性	类型
<code>()</code>	从左到右	圆括号
<code>++</code> , <code>--</code> , <code>static_cast &lt;类型&gt;()</code>	从左到右	一元(后缀)
<code>++</code> , <code>--</code> , <code>+</code> , <code>-</code>	从右到左	一元(前缀)
<code>*</code> , <code>/</code> , <code>%</code>	从左到右	乘
<code>+</code> , <code>-</code>	从左到右	加
<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	从左到右	插入/提取
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code>	从左到右	关系
<code>==</code> , <code>!=</code>	从左到右	相等
<code>?:</code>	从右到左	条件
<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>	从右到左	赋值
<code>,</code>	从左到右	逗号

图 2.15 已讨论过的所有操作符的优先级及结合性

## 2.13 计数器控制重复的本质

在一个由计数器控制的重复中,需要包括以下内容:

- (1) 一个控制变量的名称(或循环计数器);
- (2) 控制变量的初始值;
- (3) 用于检测控制变量终值的条件(亦即循环是否应继续);
- (4) 循环时,控制变量每一次改动的自增量(或自减量)。

不妨考虑如图 2.16 所示的简单程序,它的作用是打印从 1 到 10 的数字。对于第 10 行的以下声明来说:`int counter = 1;`它的作用是为控制变量(`counter`)命名,把它声明成一个整数,在内存中为其预约空间,并将它的初始值设为 1。对于要求初始化的声明来说,它实际上是一种可执行语句。在 C++ 中,对于同时还要预约内存的声明来说(如上述声明所示),它的确切名称是定义。

```

1  //Fig.2.16: fig02_16.cpp
2  //Counter - controlled repetition
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  |
10     int counter = 1;           //initialization
11
12     while ( counter <= 10 ) |  //repetition condition
13         cout << counter << endl;
14         ++counter;             //increment
15     |
16
17     return 0;
18 |

```

输出结果:

```

1
2
3
4
5
6
7
8
9
10

```

图 2.16 计数器控制的重复

`counter` 的声明和初始化还可用语句

```
int counter;
counter = 1;
```

来完成。我们可用上述两种方法完成变量的初始化。

语句

```
++counter;
```

用于每次执行循环时,都让循环计数器自增 1。while 结构中的循环条件检测控制变量的值是否小于或等于 10(这是让条件保持 true 的最后一个值)。注意即使控制变量等于 10,while 的主体也是要执行的。一旦控制变量大于 10(即 counter 变成 11),则循环中止。

为使图 2.16 的程序更简练,还可将 counter 初始化为 0,具体做法是用语句

```
while ( ++counter <= 10 )
    cout << counter << endl;
```

替换原来的 while 结构上述代码省略了一条语句,因为自增是在 while 条件检测之前,在其中直接完成的。另外,上述代码省去了围绕 while 主体的花括号,因为 while 现在只包含一条语句。不过,要采用如此简炼的方法进行编码,需要多加练习才行。另外,它会加大调试、修改和维护程序的难度。

**常见编程错误 2.12** 由于浮点值只是近似值,所以用浮点变量对计数循环进行控制,会出现不准确的计数器值,并会出现对中止条件的不准确检测。

**良好编程习惯 2.15** 用整数值控制计数循环。

**良好编程习惯 2.16** 每个控制结构主体中的语句都要进行缩进处理。

**良好编程习惯 2.17** 在每个控制结构前后都留一个空行,将其同程序的其余部分区分开。

**良好编程习惯 2.18** 嵌套级别过多,会导致程序难于理解。嵌套应通常控制在 3 级以内。

**良好编程习惯 2.19** 控制结构上下的垂直间距,以及在控制结构头部对控制结构主体的缩进,可为程序员营造一种二维外观,极大增强可读性。

## 2.14 for 重复结构

for 结构可处理计数器控制重复的所有细节。为便于大家体验 for 结构的强大功能,我们改写了图 2.16 中的程序,结果如图 2.17 所示。改写后的程序为

```
1 //Fig.2.17: fig02_17.cpp
2 //Counter - controlled repetition with the for structure
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 |
10     // Initialization, repetition condition, and incrementing
11     // are all included in the for structure header.
12
```

```

13     for ( int counter = 1; counter <= 10; counter ++ )
14         cout << counter << endl;
15
16     return 0;
17 ;

```

图 2.17 用 for 结构实现计数器控制重复

for 结构开始执行时,会声明控制变量 counter,并把它初始化为 1。随后,检测是否符合继续循环的条件:counter <= 10。由于 counter 的初始值是 1,所以条件满足,主体语句会打印出 counter 的值,亦即 1。随后,我们用 counter ++ 这个表达式使变量 counter 自增。然后,下一次循环继续条件检测开始。此时,控制变量的值应变成 2,表明尚未抵达终值,所以主体语句会再执行一遍。类似的处理会一直继续下去,直到控制变量 counter 的值自增至 11——这会导致循环继续检测失败,从而中止重复。随后,程序会继续执行位于 for 结构之后的第一条语句(就目前来说,是执行程序尾部的 return 语句)。

图 2.18 对 for 结构进行了更形象的讲解。注意 for 结构“包揽了全部事情”——它利用一个控制变量,指定了进行计数器控制的重复时需要用到的每一项。如 for 的主体包含多条语句,就必须用花括号将循环主体封闭起来。

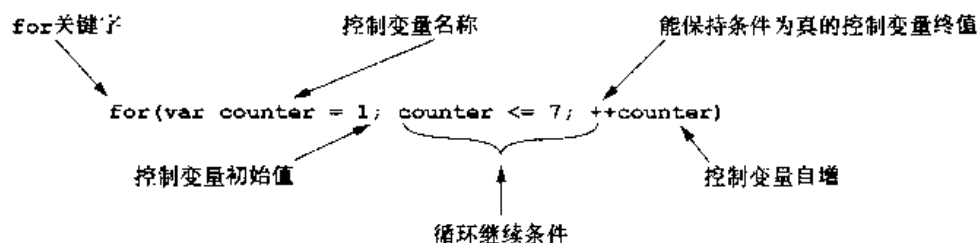


图 2.18 典型 for 结构的头部组件

注意在图 2.17 中,我们采用的循环继续条件是 counter <= 10。如不慎写成了 counter < 4,循环就只能执行 9 次。这属于一种常见的逻辑错误(值相差 1 错误)。

**常见编程错误 2.13** 假如在 while 或 for 结构的条件中使用了一个不正确的关系操作符,或者使用了一个不正确的循环计数器终值,便可能导致值相差 1 错误。

**良好编程习惯 2.20** 在 while 或 for 结构的条件中使用终值,并使用关系操作符 <=,有助于避免产生值相差 1 错误。例如,对一个用于打印从 1 到 10 的循环来说,循环继续条件应是 counter <= 10,而不应是 counter < 10(后者会产生值相差 1 错误),也不应是 counter < 11(尽管仍然是正确的)。许多程序员仍然喜欢所谓的“零基计数”。也就是说,为通过一个循环计数 10 次,先将 counter 初始化为零,再将循环继续检测条件设为 counter < 10。

for 结构的常见格式为

```

for (initialization; loopContinuationTest; increment)
    statement

```

其中,“初始化”表达式用于对循环的控制变量进行初始化;“循环继续检测”是检测循环是否应继续的一个条件(其中包含保持条件为“true”时的控制变量终值);而“自增”用于使控制变量自增。大多数情况下,for 结构都可换用一个对等的 while 结构来表达,如下所示:

```
initialization;

while (loopContinuationTest) {
    statement
    increment;
}
```

不过,此规则也有一个例外,详情参见 2.18 节。

假如 for 结构头部的初始化表达式定义了控制变量(亦即在变量名之前,指定了控制变量的类型),那么控制变量只能在 for 结构主体中使用。换句话说,在 for 结构之外,控制变量的值是未知的。对控制变量名的使用进行的限制称为变量的作用域。变量的作用域指出它可在程序中的任何地方使用。有关作用域的详情,参见第 3 章。

**常见编程错误 2.14** 假如一个 for 结构的控制变量最早是在 for 结构头部的初始化小节定义的,那么在结构的主体之后,再使用控制变量便会导致语法错误。

**可移植性提示 2.1** 在 C++ 标准中,在一个 for 结构的初始化部分声明的控制变量的作用域不同于老版本 C++ 编译器所声明的作用域。在兼容于 C++ 标准的编译器上,如果对以前采用老版本 C++ 编译器所创建的 C++ 代码进行编译,那么可能出现中断情况。有两种保守型的编程策略可用于防止此类问题的产生。一个办法是在每个 for 结构中都用不同的名称定义控制变量。另外,如果你希望在几个 for 结构中为控制变量使用相同的名称,那么在外部并且在第一个 for 循环之前定义控制变量。

有时,初始化和自增表达式也可写成一个用逗号分隔的表达式列表。在这种使用场合下,逗号操作符可保证表达式列表按“从左到右”的顺序求值。在所有 C++ 操作符中,逗号操作符的优先级是最低的。对一个逗号分隔的表达式列表来说,它的值和类型就是列表中最右边那个表达式的值和类型。逗号操作符经常在 for 结构中使用。它的主要用途是让程序员使用多个初始化表达式以及/或者多个自增表达式。例如,在一个 for 结构中,可能存在着几个控制变量,它们必须初始化和自增。

**良好编程习惯 2.21** 在 for 结构的初始化及自增部分,应尽量只用与控制变量有关的表达式。如还需对其他变量进行处理,请要么在循环之前进行(前提是它们只执行一次,比如初始化语句),要么在循环主体内进行(前提是每次重复都要执行,比如自增或自减语句)。

事实上,for 结构的 3 个表达式是可选的。如省略“循环继续检测”部分,C++ 会假定循环继续条件为“真”,所以会产生一个无限循环。什么情况下可以省略“自增”部分呢?在两种情况下都可以这样:稍后由 for 结构主体的语句计算了自增,或者根本不需要用到自增。注意出现于 for 主体尾部的自增表达式其实一条独立的语句。所以表达式

```
counter = counter + 1
counter += 1
++ counter
counter ++
```

完全等价于 for 结构的自增部分。许多程序员都喜欢采用 counter ++ 的形式,因为这样一来,自增会在循环主体执行完毕后才开始自增。后自增的形式显得更加自然。由于这里自增的

变量不必出现在一个表达式中,所以不管前自增还是后自增,效果都是一样的。注意 for 结构的两个分号是必需的。

**常见编程错误 2.15** 在 for 结构的头部语句,假如误将两个分号写成逗号,会造成语法错误。

**常见编程错误 2.16** 如果在 for 结构的头部语句的右括号之后,紧接着便加一个分号,会导致 for 结构的主体变成一条空语句。这通常会造成逻辑错误。

**软件工程知识 2.9** 有时,可特意在 for 头部之后加上一个分号。其目的是创建所谓的延迟循环。像这样带有一个空主体的 for 循环仍会正常地循环指定的次数,只是除计数之外,无任何功能。例如,你可利用一个延迟循环放慢程序的执行速度,确保屏幕输出不至于过快,避免无法阅读。

for 结构的初始化、循环继续条件及自增部分只能包含算术表达式。举个例子来说,我们可假定  $x = 2$  和  $y = 4$ 。现在,假如循环主体内没有改动  $x$  和  $y$ ,那么语句

```
for ( int j = x; j <= 4 * x * y; j -= y / x )
```

其实完全等价于语句

```
for ( int j = 2; j <= 80; j += 5 )
```

for 结构的“自增”部分也可负数(效果相当于自减,循环会倒计数)。

假如循环继续条件一开始便为“假”,那么 for 结构的主体是永远不会执行的。相反,此时会跳过 for 结构,直接执行之后的语句。

在 for 结构主体,我们频繁地使用或打印控制变量,但这其实并非必要的。更常见的一种情况是:我们只用控制变量来控制重复。在 for 结构主体内部,可能根本不会用到它。

**良好编程习惯 2.22** 尽管可在 for 循环主体中更改控制变量的值,但尽量避免这样做,因为这样可能导致不易察觉的逻辑错误。

for 结构的流程图看起来与 while 类似。图 2.19 展示的是 for 语句

```
for ( int counter = 1; counter <= 10; counter ++ )
```

```
cout << counter << endl;
```

的流程图。该流程图清楚地揭示了初始化操作只进行一次,而自增操作会在主体语句每次

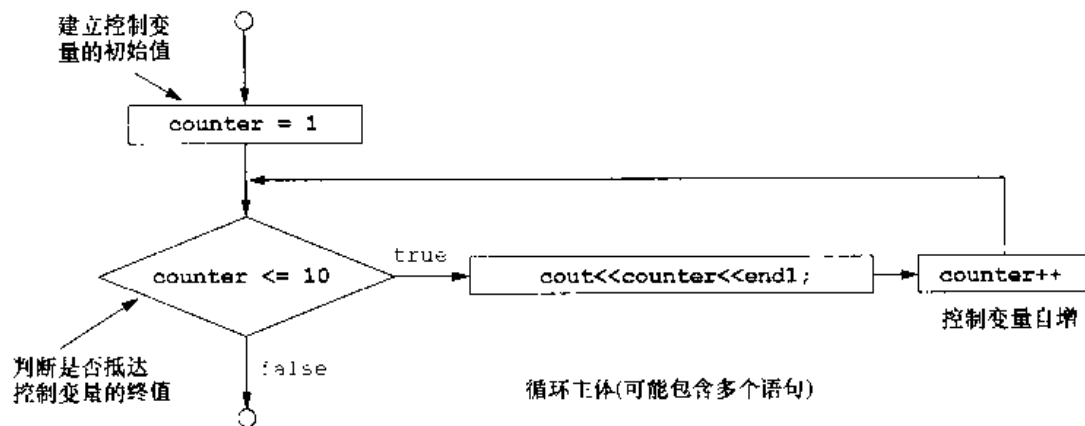


图 2.19 典型 for 重复结构的流程图

执行完之后都进行一次。再次提醒大家,在一个流程图中,除小圆圈和箭头之外,只能使用矩形和菱形符号。可想象程序员自己有一个包含了大量空 for 结构的柜子——这些结构可与其他控制结构一道堆叠或嵌套起来,从而实现一个预期的算法。然后,程序员需要用具体的行动和决定来填充这些矩形和菱形。

## 2.15 for 结构用法示例

下而这些例子展示了在 for 结构中,使控制变量变化的各种方法。在每种情况下,我们都会写下对应的 for 头部语句。注意为了让控制变量自减,关系操作符针对循环进行的改动。

a) 控制变量从 1 变化至 100,每次自增 1。

```
for ( int i = 1; i <= 100; i ++ )
```

b) 控制变量从 100 变化至 1,每次自增 -1(自减 1)。

```
for ( int i = 100; i >= 1; i -- )
```

**常见编程错误 2.17** 在一个循环的循环继续条件中,假如未能正确地使用关系操作符,以便进行倒数(比如在循环中不正确地使用  $i \leq 1$  倒数至 1),通常会造成逻辑错误,使程序运行时出现不正确的结果。

c) 控制变量从 7 变化至 77,步进 7。

```
for ( int i = 7; i <= 77; i += 7 )
```

d) 控制变量从 20 变化至 2,步进 -2。

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) 控制变量依次变化为以下一系列值:2,5,8,11,14,17,20。

```
for ( int j = 2; j <= 20; j += 3 )
```

f) 控制变量依次变化为以下一系列值:98,88,77,66,55,44,33,22,11,0。

```
for ( int j = 99; j >= 0; j -= 11 )
```

下而两个例子提供了 for 结构的简单应用。图 2.20 中的程序采用 for 结构求 2~100 之间所有偶数的总和。

```
1 //Fig.2.20; fig02_20.cpp
2 //Summation with for
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 |
10     int sum = 0;
11
12     for ( int number = 2; number <= 100; number += 2 )
13         sum += number;
14
15     cout << "Sum is " << sum << endl;
```

```

16
17     return 0;
18 }

```

输出结果:

Sum is 2550

图 2.20 用 for 求和

注意在图 2.20 中,for 结构主体实际可合并到 for 头部结构的最右边部分中,这是用逗号操作符来实现的。如下所示

```

for ( int number =2;                //initialization
     number <=100;                  //continuation condition
     sum +=number, number +=2)      //total and increment
;

```

赋值语句 `sum =0` 也可合并到 for 的初始化部分。

**良好编程习惯 2.23** 尽管 for 之前的语句以及 for 之内的语句通常都可合并到 for 的头部,但最好不要这样,因为会降低程序的可读性。

**良好编程习惯 2.24** 尽量把任何控制结构的头部限制在一行以内。

下面将用 for 结构计算复利。问题陈述如下:

某人新开了一个户头,存入 1000.00 美元,年利率为 5%。假定所有利息收入都重新存入户头,请计算并打印在为期 10 年的时间里,每一年结束时的账面金额。这些金额的计算公式为:

$$a = p(1 + r)^n$$

其中:

$p$  是最开始存入的金额(本金)

$r$  是年利率

$n$  是年数

$a$  是在第  $n$  年结束时的利滚利存款

为解决这个问题,可使用一个循环,计算 10 年中每年累加的存款金额。图 2.21 展示了具体的程序。

```

1 //Fig.2.21: fig02_21.cpp
2 //Calculating compound interest
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8
9 #include <iomanip>
10
11 using std::setw;
12 using std::setiosflags;
13 using std::setprecision;
14

```



```

15 #include <cmath>
16
17 int main()
18 {
19     double amount,           //amount on deposit
20         principal = 1000.0,   //starting principal
21         rate = .05;           //interest rate
22
23     cout << "Year" << setw( 21 )
24         << "Amount on deposit" << endl;
25
26     //set the floating-point number format
27     cout << setiosflags( ios::fixed | ios::showpoint )
28         << setprecision( 2 );
29
30     for ( int year = 1; year <= 10; year ++ ) {
31         amount = principal * pow( 1.0 + rate, year );
32         cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
33     }
34
35     return 0;
36 }

```

输出结果:

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.62
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

图 2.21 用 for 结构计算复利

for 结构将执行循环主体 10 次,使一个控制变量自 1 变化至 10,每次自增 1。C++ 没有提供一个现成的乘方操作符,所以我们采用标准库函数 pow 来做这件事情。函数 pow( $x$ ,  $y$ ) 可计算出值  $x$  的  $y$  次方。在这个例子中,代数表达式  $(1+r)^n$  被写成 pow(1 + rate, year)。其中,变量 rate 代表  $r$ ,而变量 year 代表  $n$ 。函数 pow 要求取得类型为 double 的两个参数,并返回一个 double 值。

若不包含 <cmath>,该程序将无法编译的。函数 pow 要求使用两个 double 参数。注意 year 是一个整数。<cmath> 文件包括了相关的信息,告诉编译器在调用函数之前,将 year 的值转换成一个临时性的 double 值。这些信息包括在 pow 的函数原型中。函数原型的概念将在第 3 章讲述。第 3 章还提供了对 pow 函数以及其他数学库函数的一个总结。

**常见编程错误 2.18** 如果在使用了数学库函数的程序中忘记包含 <cmath>,会导致语法错误。

注意我们将变量 `amount`、`principal` 和 `rate` 声明为 `double` 类型。这样做只是为了简化起见,因为我们打算处理带小数的美元金额,所以需要一种类型,它允许在值中使用小数。但不幸的是,这样会出现问题。在这里,不妨举一下简单的例子,它解释了假如用 `float` 或 `double` 表示美元金额,会造成什么样的麻烦(假定用 `setprecision(2)` 指定打印时采用两位精度):机器中保存的两个美元金额可能是 14.234(打印成 14.23)和 18.673(打印成 18.67)。如将这两个金额相加,内部求和应得到 32.907,打印成 32.91。所以你的打印结果可能是

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

但是,假如你自己将上述打印出来的两个数字相加,得到的总和应该是 32.90! 请务必注意这个问题!

**良好编程习惯 2.25** 不要用 `float` 或 `double` 类型的变量来执行财务计算。不精确的浮点数会造成错误,得到不正确的金额。在练习中,我们探讨了用整数执行金融计算的方法。注意:可选择由第三方厂商提供的 C++ 类库,它们能正确地执行金融计算。

注意 `for` 循环之前的输出语句

```
cout << setiosflags( ios::fixed | ios::showpoint )
      << setprecision( 2 );
```

和 `for` 循环内部的输出语句

```
cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
```

两者合并起来,以便打印变量 `year` 和 `amount` 的值,同时采用由参数化流操纵元 `setw`、`setiosflags` 和 `setprecision` 所指定的格式。其中,调用 `setw(4)` 意味着下一个输出值在打印时采用的域宽为 4。也就是说,值在打印时至少需要占用 4 个字符位置。如果要输出的值本身的宽度不到 4 个字符,那么在默认情况下,值会在字段中采用右对齐。如果要输出的值本身的宽度已不止 4 个字符,那么域宽将进行扩展,以便同整个值相容。通过调用 `setiosflags( ios::left )`,可强行规定值应该采用左对齐的方式输出。

在上述输出语句中,其他格式指出变量 `amount` 会作为一个定点值打印,它的小数点(由流操纵元 `setiosflags( ios::fixed | ios::showpoint )` 指定)将在占据 21 个字符位置(由 `setw(21)` 指定)的字段中进行右对齐。同时在小数点右侧,采用两个数位的精度(由 `setprecision(2)` 指定)。第 11 章将讨论 C++ 强大的输入/输出格式功能。我们在 `for` 循环之前的 `cout` 中放置了 `setiosflags` 和 `setprecision` 这两个流操纵元,因为这些设置在被修改之前,一直都是有效的。因此,不需要在循环的每次重复期间都应用它们。

注意: `1.0 + rate` 这一计算式,它作为一个参数出现在 `pow` 函数中,并整个包含在 `for` 语句的主体中。但事实上,该计算会在每次通过循环时都产生相同的结果,所以重复计算是颇为浪费的。

**性能提示 2.5** 不要在循环中放置值不会发生改变的表达式——但是,即使你真的这样做了,如今许多高级的优化编译器也会在其生成的机器代码中,自动将这样的表达式排除在循环之外。

**性能提示 2.6** 许多编译器都包含了优化功能,可对你写的代码进行改进,但最好一开始便

养成一次性写好代码的习惯。

为提高趣味性,可尝试一下本章练习题中的“Peter Minuit”问题。该问题演示了复利的奇迹。

## 2.16 switch 多选结构

我们已讨论了 if 单选结构以及 if/else 双选结构。有时,在一个算法中会包含一系列决定。在这些决定中,需针对它能假定的每一个常量整数值,单独测试一个变量或表达式,并据此采取不同的行动。C++ 对此提供了 switch 多选结构。

switch 结构由一系列 case(条件)标签构成,还可选择在其中包含一个 default(默认)条件。图 2.22 中的程序利用 switch 统计一次考试成绩中,各等级(采用字母得分)对应的学生人数。

```

1 //Fig. 2.22: fig02_22.cpp
2 //Counting letter grades
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     int grade, //one grade
12         aCount = 0, //number of A's
13         bCount = 0, //number of B's
14         cCount = 0, //number of C's
15         dCount = 0, //number of D's
16         fCount = 0; //number of F's
17
18     cout << "Enter the letter grades. " << endl
19          << "Enter the EOF character to end input. " << endl;
20
21     while ( ( grade = cin.get() ) != EOF ) {
22
23         switch ( grade ) { //switch nested in while
24
25             case 'A': //grade was uppercase A
26             case 'a': //or lowercase a
27                 ++aCount;
28                 break; //necessary to exit switch
29
30             case 'B': //grade was uppercase B
31             case 'b': //or lowercase b
32                 ++bCount;
33                 break;
34

```

```

35     case 'C': //grade was uppercase C
36     case 'c': //or lowercase c
37         ++cCount;
38         break;
39
40     case 'D': //grade was uppercase D
41     case 'd': //or lowercase d
42         ++dCount;
43         break;
44
45     case 'F': //grade was uppercase F
46     case 'f': //or lowercase f
47         ++fCount;
48         break;
49
50     case '\n': //ignore newlines,
51     case '\t': //tabs,
52     case ' ': //and spaces in input
53         break;
54
55     default: //catch all other characters
56         cout << "Incorrect letter grade entered. "
57             << " Enter a new grade." << endl;
58         break; //optional
59     |
60 }
61
62 cout << "\n\nTotals for each letter grade are: "
63     << "\nA: " << aCount
64     << "\nB: " << bCount
65     << "\nC: " << cCount
66     << "\nD: " << dCount
67     << "\nF: " << fCount << endl;
68
69 return 0;
70 |

```

#### 输出结果:

```

Enter the letter grades.
Enter the EOF Character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D

```

A  
b

Totals for each letter grade are:

A:3  
B:2  
C:3  
D:2  
F:1

图 2.22 switch 用法示例

在程序中,用户需要输入代表一个班平均成绩的字母。while 结构的头部

```
while ( ( grade = cin.get() ) != EOF )
```

先执行括号里的赋值语句( `grade = cin.get()` )。 `cin.get()` 函数可从键盘读回一个字符,并将其保存在整数变量 `grade` 中。 `cin.get()` 中使用的小数点符号将在第 6 章“类和数据抽象”中进行解释。字符通常保存在类型为 `char` 的变量中;然而,C++ 一项重要的特性便是字符可保存到任何整数数据类型中,因为它们在计算机里是用 1 个字节的整数表示的。因此,我们可将字符看待成一个整数或者一个字符,具体由它的用途决定。比如语句

```
cout << "The character('a') has the value"
      << static_cast<int>('a') << endl;
```

其作用是打印字符 `a` 及其整数值,如下所示

```
The character (a) has the value 97
```

其中,整数 97 是字符在计算机中的数值表达形式。今天许多计算机都使用 ASCII(美国标准信息交换码)字符集。97 代表小写字母 'a'。ASCII 字符及其十进制值的列表参考本书附录。

赋值语句作为一个整体拥有赋给 = 符号左侧的变量的值。因此,赋值语句 `grade = cin.get()` 的值等于由 `cin.get()` 返回并赋给变量 `grade` 的值。

正由于赋值语句有自己的值,所以特别适用于将几个变量初始化为相同的值。例如

```
a = b = c = 0;
```

它首先对赋值语句 `c = 0` 进行求值(因为 = 操作符是从右到左结合的)。然后,变量 `b` 得到 `c = 0` 的赋值结果(即 0)。然后,变量 `a` 得到 `b = (c = 0)` 的赋值结果(还是 0)。在程序中,`grade = cin.get()` 的赋值结果会同 EOF 的值比较(EOF 代表“end-of-file”,是用于标记文件结束的一个符号)。我们用 EOF(它的值通常是 -1)作为标记。然而,你不可直接键入 -1 值,或键入 EOF 这 3 个字母作为标记。相反,你应按一个由具体系统而决定的组合键,它对应于“文件结束”,或者表示“我没有更多的数据需要输入了”。EOF 是一个象征性的符号常量,定义于 `<iostream>` 头文件中。假如赋给 `grade` 的值等于 EOF,那么程序就会中止。我们在这个程序中之所以选择将字符表示成 `int`,是由于 EOF 本身是一个整数值(再一次提醒你,它通常等于 -1)。

**可移植性提示 2.2** 用于输入“文件结束”(EOF)的组合键视具体的系统而定。

**可移植性提示 2.3** 如检测符号常量 EOF,而不是检测 -1,会使程序更易移植。ANSI 标准规定,EOF 取一个负的整数值(但不一定是 -1)。因此,EOF 在不同的系统上取值可能不同。

在 UNIX 系统以及其他许多系统中,“文件结束”输入是在单独一行上按组合键

```
<Ctrl-d>
```

它的意思是同时按 Ctrl 键和 d 键。在其他系统上(比如 DEC 的 VAX VMS 或 Microsoft MS-DOS),文件结束可通过以下组合键

<Ctrl-z>

来输入。注意:在某些情况下,可能须要在按了上述两个组合键之后,再按一次回车。

用户用键盘输入成绩。按下回车键后,字符会由 `cin.get()` 读入,而且采取每次读入一个字符的方式。假如输入的字符不是 EOF,那么进入 switch 结构。在 switch 关键字之后,是放在括号中的变量名 `grade`。这叫做控制表达式。该表达式的值会与每一个 case 标签进行比较。假定用户输入字母 C 作为成绩,那么 C 自动与 switch 中的每一个 case 进行比较。假如找到一个匹配项(case 'C':),则执行用于那个 case 的语句。对于字母 C, `cCount` 会自增 1,而 switch 结构会随着 break 语句立刻退出。注意同其他控制结构不同,不需用花括号将一个多语句的 case 封闭起来。

break 语句会导致程序控制转到 switch 结构的第一条语句。之所以要使用 break 语句,是由于不这样做,switch 语句中的各个 case 会一起运行。假如 switch 结构中没有使用任何 break,那么每次在结构中找到一个匹配项时,用于剩下的所有 case 的语句都会执行(假如要为几个 case 采取相同的行动,比如在图 2.22 的程序中,这一特性有时也非常有用)。假如没有任何找到匹配,则执行 default 条件,它会打印一条错误提示消息。

每个 case 都可有一个或多个行动。switch 结构同其他所有结构的区别在于,不需要用花括号来括住 switch 结构中一个 case 内的多个行动。图 2.23 展示了常规的 switch 多选结构(在每个 case 中都使用一个 break)。

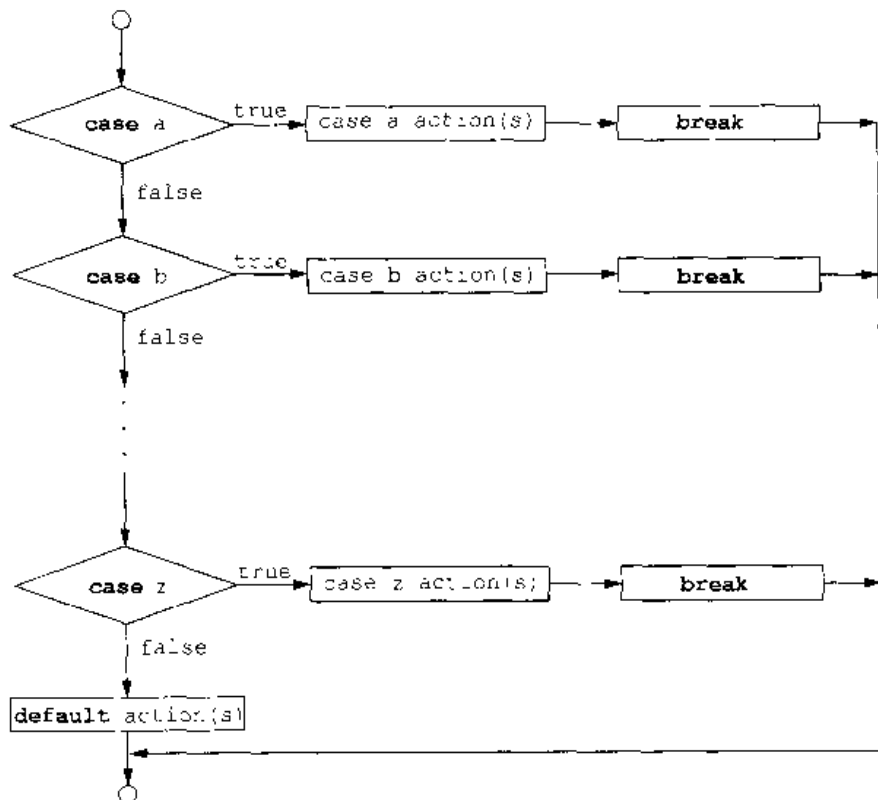


图 2.23 使用 break 的 switch 多选结构

流程图清楚地说明,每个 case 末尾的 break 语句都会导致控制流程立即退出 switch 结构。同样地,注意(除小圆圈和箭头之外)流程图只包含了矩形和菱形符号。假设程序员能

访问由多个空 switch 结构构成的一个大柜子。程序员可根据需要堆叠和嵌套尽可能多的 switch 结构,以构成某种算法的控制流程的结构化实现。同样地,需要在矩形和菱形中填充一系列行动和决定,以便同算法配合。嵌套的控制结构是最常见的,但偶尔也能在一个程序中发现嵌套的 switch 结构。

**常见编程错误 2.19** 如忘了在 switch 结构中放置一条必要的 break 语句,会造成逻辑错误。

**常见编程错误 2.20** 如果在“case”同 switch 结构中检测的一个整数值之间遗漏了空格,可能会导致逻辑错误。例如,若写成 case3:,而不是 case 3:,将创建一个无用的标签(详情参见第 18 章)。也就是说,假如 switch 的控制表达式的值是 3,switch 结构就无法采取恰当的行动。

**良好编程习惯 2.26** 无论如何都应在 switch 语句中提供 default 条件。在无 default 条件的 switch 语句中,没有明确进行检测的情况会被忽略。如包括 default 条件,会使程序员关注对例外情况的需求。某些情况下不需要进行 default 处理。尽管 switch 结构中的 case 语句和 default 语句可按任意顺序排列,但作为一个良好的习惯,应将 default 从句放在最后。

**良好编程习惯 2.27** 在 switch 结构中,假如 default 从句列于最后,就不需要为它使用 break 语句。有的程序员要包括 break 的目的是为了更有条理,以及与其他 case 语句对应。

图 2.22 的 switch 结构中,第 50~53 行

```
case '\n':
case '\t':
case ' ':
    break
```

会导致程序跳过换行符、制表位和空格字符。一次读一个字符有时会导致问题。为了让程序读入字符,必须通过在键盘上按回车键的方式,把这些字符送入计算机。但是一旦按回车键,会在希望处理的字符之后多加一个换行符。通常,必须专门对这个换行符进行处理,以便程序正常工作。通过在我们的 switch 结构中包括上述 3 个 case,就可避免每次输入一个换行符、制表位或空格时,都由 default 语句打印错误提示消息。

**常见编程错误 2.21** 每次读取字符时,假如不对输入的换行和其他“空白”字符进行处理,会导致逻辑错误。

注意假如连写几个 case 标签(比如图 2.22 中的 case 'D'; case 'd'),意味着针对每一种情况,都采取相同的一系列操作。

使用 switch 结构时,记住它只能用于测试常量整型表达式——即字符常量和整型常量的任意组合,但求值结果为一个常量整型值。整型常量表示为用一对单引号封闭起来的特定字符,比如 'A';而整型常量就是整型值。

在本书中,开始学习面向对象的编程时,我们会展示一个更好的方式实现 switch 逻辑。届时,我们会使用一项名为多态性的技术创建程序。与直接采用 switch 逻辑的程序相比,这样的程序通常更清晰、更精确、更易维护和更易扩展。

对于类似于 C++ 的易于移植语言来说,肯定具有灵活的数据类型长度。不同的应用程序可能要求不同长度的整数。C++ 提供了几种数据类型来表示整数,每种类型的整数取值范围取决于特定的计算机硬件。除 int 和 char 之外,C++ 还提供了 short(short int 的简称,即

短整型)和 long 类型(long int 的简称,即长整型)。short 整型的最小取值范围是 -32 768 ~ 32 767。对于绝大多数整数计算,long 整型已经足够了。long 整型的最小取值范围是 -2 147 483 648 ~ 2 147 483 647。在大多数计算机上,int 不是等价于 short,就是等价于 long。int 的取值范围至少应与 short 整型相同,但不会大于 long 整型的取值范围。数据类型 char 可用于表示计算机字符集中的任何字符,也可用于表示小整数。

**可移植性提示 2.4** 由于 int 的长度在不同系统上是不同的,所以要处理的整数如果超过了 -32 768 ~ 32 767 这一范围,或者希望能在多种不同的计算机系统上运行程序,就应该使用 long 整数。

**性能提示 2.7** 在性能要求较高的环境中(内存宝贵,或要求加快执行速度),尽量使用长度较短的整数。

**性能提示 2.8** 使用长度较短的整数,机器负责处理它们的指令却不如处理自然长度的整数有效(例如一定要为它们加上符号扩展)时,程序运行速度反而会减慢。

**常见编程错误 2.22** 如在 switch 结构中提供相同的标签,会造成语法错误。

## 2.17 do/while 重复结构

do/while 重复结构类似于 while 结构。不过在 while 结构中,循环继续条件会在循环起始处执行检测,在循环主体执行之前。相反,do/while 结构会在执行了循环主体之后,再检测循环继续条件。换句话说,循环主体至少会执行一次。do/while 中止后,会自 while 从句之后的第一条语句开始执行。注意假如主体中只有一条语句,那么不必在 do/while 结构中使用花括号;然而,人们通常都总是加上花括号,以免混淆 while 和 do/while 结构。比如

```
while (条件)
```

通常被认为是 while 结构的头部。假如一个 do/while 结构没有用花括号封闭单一语句主体,所以结构

```
do
    语句
while (条件);
```

会使人觉得迷惑。最后一行——while(条件);可能会被读者误解为一个包含了空语句的 while 结构。因此,为避免混淆,含有单一语句的 do/while 结构通常应写成以下形式

```
do {
    语句
} while (条件);
```

**良好编程习惯 2.28** 有的程序员习惯 While 结构中包含花括号——即使括号并无实际用途。这样有助于区分 while 结构和只包含了一条语句的 do/while 结构。

**常见编程错误 2.23** 在 while、for 或 do/while 结构中,假如循环继续条件永远变不成 false,就产生无限循环。为避免这个问题,一定要在循环头或主体的某个地方更改条件值,使条件最终能变成 false。



图 2.24 中的程序利用一个 do/while 重复结构打印从 1 到 10 的整数。注意,在检测是否继续循环时,我们让控制变量 counter 进行前自增。还要注意,我们使用花括号将 do/while 结构的单语句主体封闭起来。

```

1 //Fig.2.24; fig02_24.cpp
2 //Using the do/while repetition structure
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 |
10     int counter = 1;
11
12     do {
13         cout << counter << " ";
14     } while ( ++counter <= 10 );
15
16     cout << endl;
17
18     return 0;
19 |

```

输出结果:

```
1 2 3 4 5 6 7 8 9 10
```

图 2.24 do/while 结构用法示例

do/while 结构的流程图如图 2.25 所示。根据这个流程图,可以清楚地看出,只有至少采取了一项行动(矩形)之后,才会对决定循环是否继续的条件进行检测。同样地,每个流程图只能包含矩形和菱形符号(小圆圈和箭头除外)。再比如,你可想象程序员面前摆着一个非常深的柜子,其中包含了一系列空的 do/while 结构——数量视具体情况而定,它们可堆叠在其他控制结构上,或与它们嵌套使用,以便以结构化的方式,实现具体算法的控制流程。同样地,稍后可分别用行动和决定内容填充空白的矩形和菱形,保证既定算法的顺利实现。

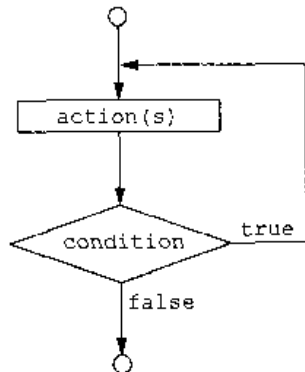


图 2.25 do/while 反复结构的流程图

## 2.18 break 和 continue 语句

break 和 continue 语句用于更改控制流程。break 语句在 while、for、do/while 或 switch 结构中执行时,会立即退出那个结构。程序会从结构之后的第一条语句开始继续执行。break 最常见的用法是提前退出一个循环,或者跳过 switch 结构剩余的部分(如图 2.22 所示)。在图 2.26 中,我们展示了在一个 for 重复结构中的 break 语句。if 结构检测到 x 变成 5 时,会执行 break。这样会中止 for 语句,程序将转到 for 之后的 cout 处继续执行。循环只完整执行 4 次。

```
1 //Fig. 2.26; fig02_26.cpp
2 //Using the break statement in a for structure
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main() {
9     |
10    //x declared here so it can be used after the loop
11    int x;
12
13    for ( x = 1; x <= 10; x++ ) {
14
15        if ( x == 5 )
16            break;    //break loop only if x is 5
17
18        cout << x << " ";
19    }
20
21    cout << " \nBroke out of loop at x of" << x << endl;
22    return 0;
23 }
```

输出结果:

```
1 2 3 4
Broke out of loop at x of 5
```

图 2.26 在 for 结构中使用 break 语句

注意在这个程序中,控制变量 x 是在 for 结构头外部定义的。这是由于我们不仅打算在循环主体中使用控制变量,还打算在循环结束执行之后使用。

continue 语句在 while、for 或 do/while 结构中执行时,会跳过此类结构的剩余部分,继续进行下一次循环。在 while 和 do/while 结构中,循环继续检测会在 continue 语句执行之后马上进行。在 for 结构中,会执行自增表达式,然后对循环继续条件进行检测。指出,大多数情况下 while 结构,用于代表 for 结构。但 continue 语句后 while 结构中的自增表达式例外。它在检测循环继续条件之前,不会执行自增,所以 while 不以相同的方式像 for 那样执行。图

2.27 在 for 结构中使用 continue 语句跳过结构中的输出语句, 直接开始下一轮循环。

```

1 //Fig. 2.27: fig02_07.cpp
2 //Using the continue statement in a for structure
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     for ( int x = 1; x <= 10; x++ ) {
11
12         if ( x == 5 )
13             continue; //skip remaining code in loop
14                        //only if x is 5
15
16         cout << x << " ";
17     }
18
19     cout << "\nUsed continue to skip printing the value 5"
20         << endl;
21     return 0;
22 }
```

输出结果:

```

1 2 3 4 5 6 7 8 9 10
Used continue to skip printing the value 5
```

图 2.27 在 for 结构中使用 continue 语句

**良好编程习惯 2.29** 有的程序员觉得 break 和 continue 违背了结构化编程准则。由于这些语句的效果可通过后文介绍的结构化编程技术实现, 所以他们不使用 break 和 continue。

**性能提示 2.9** 如运用得当, break 和 continue 语句的速度要比后文介绍的结构化编程技术快。

**软件工程知识 2.10** 在实现高质量的软件工程和获得最佳的性能之间, 总难取得平衡。往往会顾此失彼。

## 2.19 逻辑操作符

迄今为止, 我们学过的大多数条件都是简单条件, 比如 `counter <= 10`, `total > 1000` 以及 `number != sentinelValue` 等等。在表达这些条件的时候, 我们使用了关系操作符 `>`, `<`, `>=` 和 `<=`, 以及相等性操作符 `==` 和 `!=`。做出的每项决定都完全是以一个条件为基础的。假如检测多个条件才能做出一项决定, 我们需要在单独的语句中进行检测, 或在嵌套的 if 或 if/else 结构中进行。

C++ 提供了一些逻辑操作符, 可用于将简单条件合并到一起, 从而构成更复杂的条件。

这些逻辑操作符包括 && (逻辑 AND)、|| (逻辑 OR) 以及 ! (逻辑 NOT, 也叫做逻辑非)。下面分别举例说明。

假如我们想保证只有在两个条件都为 true 的前提下, 才能选择一个特定的执行路径, 就可像下面这样使用逻辑操作符 &&

```
if ( gender == 1 && age >= 65 )
    ++seniorFemales;
```

if 语句实际包含了两个简单条件。其中, 通过对 `gender == 1` 求值, 可判断一个人是否为女性; 而通过对 `age >= 65` 求值, 可判断一个人是否为年长的市民。首先求值的是 && 操作符左边的简单条件, 因为 `==` 具有比 && 更高的优先级。如有必要, 接下来会求值 && 操作符右边的简单条件, 因为 `>=` 具有比 && 更高的优先级 (如同我们稍后会讲到的, 只有在逻辑 AND 表达式左侧求值结果为 true 的前提下, 才会接着对右侧进行求值)。随后, if 语句会对以下合成条件

```
gender == 1 && age >= 65
```

进行判断。只有在两个简单条件都为 true 的前提下, 上述合成条件才会为 true。最后, 假如这个合成条件真的为 true, 那么 `seniorFemales` 的计数会自增 1。假如两个简单条件有一个或者全部为 false, 那么程序跳过自增运算, 接着执行 if 之后的下一条语句。加上括号后

```
( gender == 1 ) && ( age >= 65 )
```

上述合成条件显然更易理解。

**常见编程错误 2.24** 尽管从数学上看, 像  $3 < x < 7$  这样的表达式是完全正确的, 但在 C++ 里是绝对不允许的。应该写成:  $(3 < x \ \&\& \ x < 7)$ 。

图 2.28 总结了 && 操作符。该表针对表达式 1 和表达式 2, 列出了总共 4 种可能的真假组合。注意像这样的表格其实是大有“来历”的, 它们甚至有一个专门的学名, 叫作“真值表”(Truth Table)。对任何表达式来说, 只要它使用了关系操作符、相等性操作符以及/或者逻辑操作符, C++ 最终都会为其求出真或假的结果。

表达式 1	表达式 2	表达式 1 && 表达式 2
假	假	假
假	真	假
真	假	假
真	真	真

图 2.28 && (逻辑 AND) 操作符的真值表

**可移植性提示 2.5** 为兼容于 C++ 标准的早期版本, bool 值“true”(真)也可表示为任何非零的值; 而 bool 值“false”(假)也可表示为值 0。

接下来, 看看 || 操作符 (又称为“逻辑 OR”、“逻辑或”操作符) 的情况。在一个程序中的某个位置, 假定我们希望在采取一个特定的行动之前, 两个条件之一或者全部必须为 true, 便可考虑采用 || 操作符。如下所示

```
if ( semesterAverage >= 90 || finalExam >= 90 )
    count << "Student grade is A" << endl;
```

上述条件也是由两个简单条件合并而成。其中, `semesterAverage >= 90` 条件来说, 它用于判

断学生是否由于在学期中一贯的良好表现,所以一门课应该得到“A”;对 `finalExam >= 90` 用于判断是否由于学生的期末成绩高于或等于 90 分,所以一门课应该得到“A”。`if` 语句对合并的条件

```
semesterAverage >= 90 || finalExam >= 90
```

进行考察。如满足其中任何一个条件,或两个条件全满足,学生就可得到一个“A”。注意,只有在上述两个简单条件都为 `false` 时,才不会打印出“Student grade is A”消息。图 2.29 是逻辑 OR 操作符(`||`)的真值表。

表达式 1	表达式 2	表达式 1    表达式 2
假	假	假
假	真	真
真	假	真
真	真	真

图 2.29 `||` (逻辑 OR) 操作符的真值表

`&&` 操作符的优先级要比 `||` 操作符稍高一些,两个操作符的结合顺序都是从左向右进行的。对于包含 `&&` 或 `||` 操作符的表达式来说,一旦能确定整个表达式的真假,就会立即完成求值。也就是说,表达式

```
gender == 1 && age >= 65
```

中,假如 `gender` 不等于 1 (即整个表达式为 `false`),它会立即中止;假如 `gender` 等于 1,它则会继续(也就是说,假如条件 `age >= 65` 为 `true`,整个表达式仍然是 `true`)。

**常见编程错误 2.25** 在使用了 `&&` 操作符的表达式中,可能有一个条件——我们把这种条件称为“依赖条件”——要求在另一个条件为 `true` 的前提下,才有必要对其进行求值。在这种情况下,依赖条件应放在另一个条件之后,否则会出错。

**性能提示 2.10** 在使用了 `&&` 操作符的表达式中,假如不同的条件是相互独立的,那么将最有可能成为 `false` 的条件放在最左边。在使用了 `||` 操作符的表达式中,应把最有可能成为 `true` 的条件放在最左边。这样可缩短程序的执行时间。

C++ 还提供了 `!` (逻辑非、逻辑否定)操作符,用于逆转条件的含义。不同于 `&&` 和 `||` 的是, `!` 操作符只要求取得一个操作数;而前面两个操作符都要同时取得两个操作数。换言之, `&&` 和 `||` 均为二元操作符, `!` 却是一元操作符。逻辑非操作符必须放在打算逆转的条件之前。通常,假如我们希望在一个原始条件(没有逻辑非操作符)为 `false` 的前提下选择一个执行路径,便可考虑采用这种操作符。如下所示

```
if ( ! ( grade == sentinelValue ) )
    cout << "The next grade is " << grade << endl;
```

注意 `grade == sentinelValue` 条件两边的圆括号是必需的! 因为逻辑非操作符 `!` 的优先级高于相等性操作符。在图 2.30 中,我们总结了逻辑非操作符的真值表。

通常,假如想避免使用逻辑非操作符,可换用其他方式来陈述要检测的条件——要求使用一个合适的关系或相等操作符。例如,上述语句可改写为

```
if ( grade != sentinelValue )
    cout << "The next grade is " << grade << endl;
```

正是类似的灵活性,程序员才可按自己的意愿,以更自然、更方便的形式表达条件

表达式	! 表达式
假	真
真	假

图 2.30 ! (逻辑非)操作符的真值表

图 2.31 展示了已讨论过的所有 C++ 操作符的优先级和结合性。操作符从上到下按优先级降序排列。

操作符	结合性	类型
()	从左到右	括号
++, --, static_cast < 类型 > ()	从左到右	一元(后缀)
++, --, +, -	从右到左	一元(前缀)
*, /, %	从左到右	乘法
+, -	从左到右	加法
<<, >>	从左到右	插入/提取
<, <=, >, >=	从左到右	关系
==, !=	从左到右	相等性
&&	从左到右	逻辑 AND
	从左到右	逻辑 OR
?:	从右到左	条件
=, +=, -=, *=, /=, %=	从右到左	赋值
,	从左到右	逗号

图 2.31 操作符优先级和结合性

## 2.20 混淆相等性操作符(==)和赋值操作符(=)

无论水平高低、经验多寡,只要是 C++ 程序员,都容易犯这样的错误——该用 = 的地方用了 ==;该用 == 的地方,却又用了 =。因此,我们觉得有必要单列一节重点讨论。最糟糕的是,类似的误用不会导致系统报告语法错误。相反地,程序会忽略这些错误,并正常完成编译,而且往往能正常执行完毕。但由于隐藏了逻辑性错误,所以往往最终得到的不正确的结果。

在 C++ 中,有两处容易出现类似错误。首先是生成一个值,以便在任何控制结构的决定部分使用这个值的任何一个表达式。如值为 0,它会被视为 false;如值不为零,则被视为 true。其次是通过 C++ 赋值而生成的一个值,亦即赋给赋值操作符左侧变量的一个值。例如,假设我们本来想写代码

```
if ( paycode == 4 )
    cout << "You get a bounus!" << endl;
```

却不慎写为

```
if ( paycode = 4 )
    cout << "You get a bounus!" << endl;
```

第一个 if 语句是正确的,它能为 paycode(工资代码)是 4 的人发放奖金(You get a bonus)。但是,第二个 if 语句(有错的语句)却会把 if 条件中的赋值表达式求值为常量 4。由于所有非零的值都被解释为 true,所以这个 if 语句的条件都是 true,那个走运的人总能拿到奖金,而无视其实际的 paycode 等于多少!甚至还有更糟的,原意只是检查一下工资代码,现在却变成了主动篡改这个代码!

**常见编程错误 2.26** 无论用 == 赋值,还是用 = 判断是否相等,都属于典型的逻辑错误,但不会被视为语法错误——歧义因此而产生。

**测试和调试提示 2.1** 程序员习惯于写一些类似于 `x == 7` 的条件表达式,把变量名放到左边,把常量放到右边。但是,一种更好的做法却是逆转这个顺序,把常量放到左边,把变量名放到右边(比如 `7 == x`)。这样一来,一旦不慎将 `==` 写成了 `=`,编译器立刻会报告错误。以 `7 = x` 为例,编译器看到这样的写法,便会正确判断它是一个语法错误,因为在赋值语句中,不可以将一个变量赋给一个常数,不可以将常数放在赋值操作符的左边。这样至少可以防止运行时逻辑错误程序干扰。

说专业点,我们认为变量名是“左值”或 lvalue(left value),因为它们通常放在赋值操作符的左边。相反,常量是“右值”或 rvalue(right value),因为它们只能在赋值操作符的右边使用。尽管左值有时可跑到右边来,变成右值,但右值绝对不可以变成左值!

某些情况下,如在本该用 `=` 时用了 `==`,也会带来令人不快的结果。例如,如果程序员想把一个值赋给一个变量,只想使用语句

```
x = 1;
```

但不慎写为

```
x == 1;
```

从语法上看,这样写没有丝毫问题,只是编译器不会去赋值,而是去判断条件表达式的值。若 `x` 等于 1,则条件为 true,表达式返回 true 值;如 `x` 不等于 1,条件为 false,表达式会返回 false 值。不管返回值是什么,由于没有赋值操作符,所以返回的值会被丢弃。同时,`x` 的值未见任何变化,从而造成了一个非常典型的运行时逻辑错误。而且不幸的是,没有任何方便的技巧可帮你快速解决这样的问题。

**测试和调试提示 2.2** 用文本编辑器查找程序中所有的 `=`,查实各处使用的操作符是否正确。

## 2.21 结构化编程小结

建筑师设计建筑物时,必须综合运用自己这个行业历年积累下来的各种知识与经验。程序员设计程序时,同样如此。不过,这个行业要比建筑业年青得多,所以积累下来的知识与经验相对较少。通过以前的学习,我们已知道同非结构化程序相比,结构化编程产生的程序更容易理解,所以更易测试、调试、修改,而且更不容易出错。

图 2.32 总结了 C++ 控制结构。其中,那些小圆圈用于表示每种结构的单一入口和单一出口。假如任意连接单独的流程图符号,最终有可能导致非结构化的程序。因此,程序专家

决定把流程图符号合并到一起,构成一个有限的控制结构集。同时只需通过两种简单的方式来正确地合并控制结构,最终就能构建出一个理想的结构化程序。

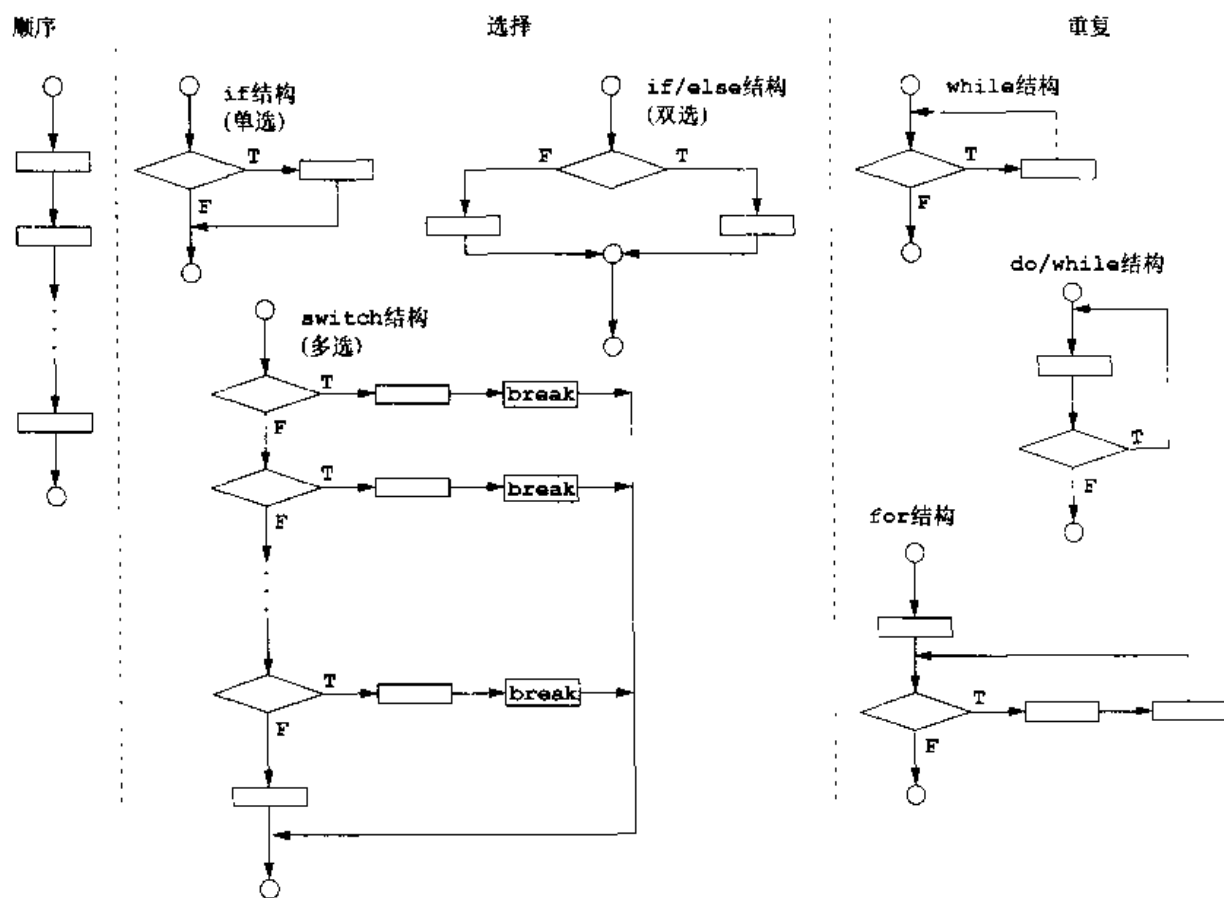


图 2.32 C++ 的单入/单出顺序、选择和重复结构

为简化起见,设计者只采用了单入/单出控制结构。换言之,每个控制结构都只有一条路可以进入,也只有一条路可以退出。按顺序连接不同的控制结构来构建结构化程序的方式最为简单:把控制结构的出口同下一个控制结构的入口连起来就可以了。也就是说,各控制结构在程序中只是逐个地放在一起——我们把这种情况称为“控制结构的堆叠”。结构化程序的构建规则也允许进行控制结构的嵌套。

图 2.33 展示了正确构建结构化程序的规则。这些规则假定用流程图中的矩形符号标记一项行动,包括输入和输出等。此外还假定从最简流程图开始(见图 2.34)。

#### 结构化程序构建规则

- (1) 从如图 2.34 所示的那个“最简流程图”开始
- (2) 任何矩形(行动)都可被替换成两个顺序的矩形(行动)
- (3) 任何矩形(行动)都可被替换成控制结构(顺序,if,if/else,switch,while,do/while 或者 for)
- (4) 规则(2)和(3)可根据自己的需要随意应用,而且可按任意顺序来应用

图 2.33 搭建结构化程序的规则



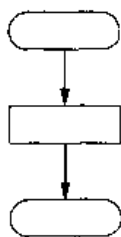


图 2.34 最简流程图

如遵循图 2.33 的规则,结果总会在结构化流程图中表现出一个清爽的、逐块叠加起来的程序结构。例如,假如不断地把规则(2)应用于最简流程图,便会产生一个结构化流程图,其中包含了一系列按顺序排列起来的矩形,如图 2.35 所示。注意由于规则(2)产生一系列控制结构的“堆栈”,所以我们有时也把规则(2)称为堆栈规则或堆叠规则。

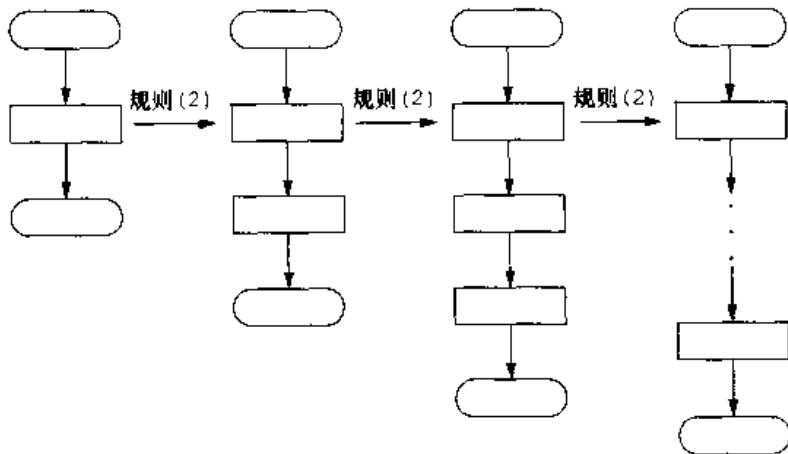


图 2.35 把图 2.33 的规则(2)反复应用于最简流程图

规则(3)称为嵌套规则。如不断地把规则(3)应用于最简流程图,会产生一个嵌套的控制结构。例如,在图 2.36 中,首先用一个双选(if/else)结构来替代最简流程图中的矩形。随后,再次将规则(3)应用于双选结构中的两个矩形,用双选结构来替换掉每一个矩形。围绕每个双选结构显示的虚线框则对应于已在最简流程图中被取代的矩形。

规则(4)可生成更大、更复杂的以及嵌套更深的结构。通过应用图 2.33 的规则,可构成所有可能的结构化流程图,所以也能产生所有可能的结构化程序。

结构化编程的好处在于,我们只需使用 7 种简单的单入/单出结构,然后只需用两种简单的方法把它们“组装”起来,最终便能构成一个可满足任意要求的程序。图 2.37 展示了一系列堆叠起来的构建单元,它们通过应用规则(2)而产生;以及一系列嵌套的构建单元,它们通过规则(3)而产生。该图还展示了构建单元相互重叠的情况,它们不可在结构化流程图中表现出来(因为已经取消了 goto 语句)。

假如坚持遵循图 2.33 的规则,不可能得到一个非结构化的流程图(如图 2.38 所示)。假如不能确定一张特定的流程图是否结构化,可逆向应用图 2.33 列举的规则,看看是否能把流程图简化至最简形式。如最后能还原为最简流程图,表明原来的流程图是结构化的;否则就是非结构化的。

结构化主要为了简化编程。Bohm 和 Jacopini 使我们理解到只有 3 种控制才是必需的:

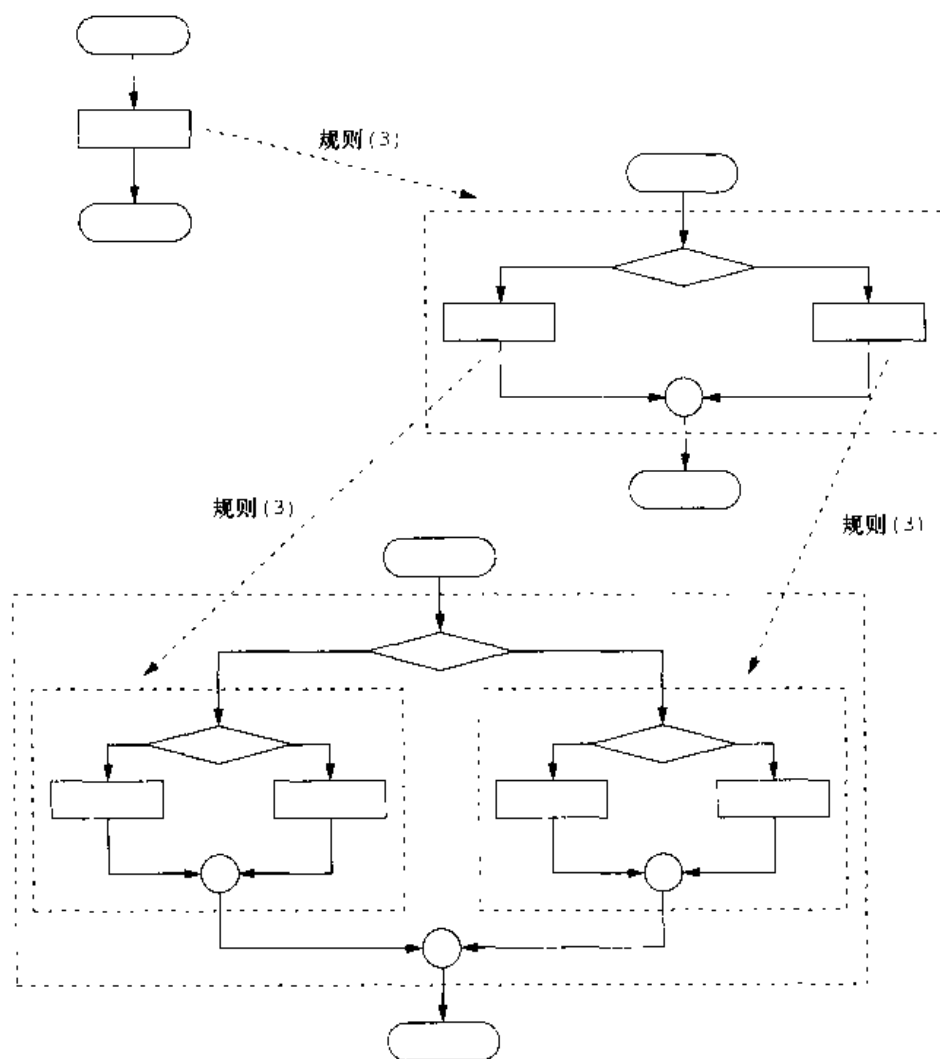


图 2.36 把图 2.33 的规则(3)应用于最简流程图

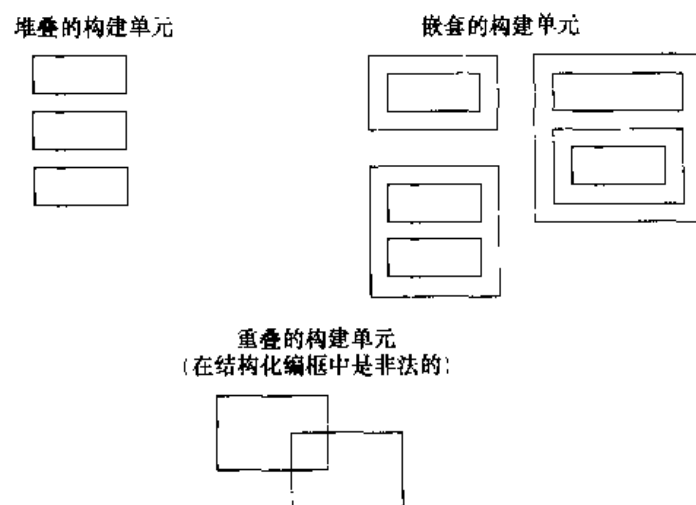


图 2.37 堆叠、嵌套和重叠的构建单元

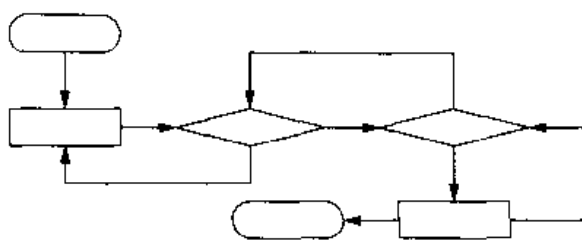


图 2.38 非结构化编程的流程图

- 顺序;
- 选择;
- 重复。

其中,顺序结构最简单。可通过下述方式之一选择:

- f 结构(单选);
- if/else 结构(双选);
- switch 结构(多选)。

其实,我们很容易证明最简单的 if 结构亦足以实现任何形式的选择。也就是说,用 if/else 和 switch 结构能做到的事,用 if 结构一样能做到(尽管条理可能有些不清晰,效率也不够高)。

可用下述方式之一实现重复:

- while 结构;
- do/while 结构;
- for 结构。

同样很容易证明,仅仅用 while 结构,亦足以实现任何形式的重复。也就是说,用其他循环结构能做到的事情,用 while 结构也能做到(尽管可能没什么条理)。

通过组合这些结果,我们总结出 C++ 程序中需要的任何形式的控制都可通过以下形式来表达。

- 顺序;
- if 结构(用于选择);
- while 结构(用于重复)。

同时,所有这些控制结构只需选择两种方式之一进行组合:堆叠和嵌套。总之,结构化机制可极大简化编程,是进行快速应用程序开发的好助手!

本章讨论了如何基于包含了行动及决定的控制结构构建程序。第 3 章将介绍另一种程序结构单元,即函数。我们将学习如何合并不同函数(每个函数又可包含多种控制结构),构建出大型的程序。我们随后会讨论函数对于软件的可重用性提供了哪些好处。第 6 章还会介绍 C++ 的另一种程序结构单元类。我们将在类的基础上创建对象,并讲述面向对象编程的概念。我们将在后面的“对象思想”小节中,提出一个实际问题,让读者思考如何利用面向对象的设计技术来解决它,从而继续讨论对象。

## 2.22 【可选案例分析】对象思想:标识问题所牵涉的类

现在,我们将开始这个可选的、面向对象的设计与实现案例分析。本章和以后几章末尾的“对象思想”小节将通过分析一个电梯模拟案例,引导你逐渐地进入面向对象的领域。该案例分析会指导你获得一套真实的、循序渐进的、完整的设计与实现体验。第2~5章将使用UML完成面向对象设计(OOD)涉及的各个步骤。第6、第7和第9章,我们将借助C++语言,采用面向对象编程(OOP)技术,实现该电梯模拟程序。这个案例分析所牵涉到的所有问题最终都会得到圆满解决。事实上,这并不是一个练习;而是非常直观的学习体验,而且最后还会逐行讲解完整的C++代码。通过该案例分析,你会逐渐习惯工业界需要解决的各种实际问题。希望你喜欢这样的体验。

### 2.22.1 问题陈述

一公司打算建造一幢两层的建筑物,并为它装一部电梯。公司想让你用C++开发一个面向对象的模拟程序,对电梯的运行进行建模,判断它是否能满足他们的要求。

你的模拟程序应包括一个时钟,它的初始值设为零(以秒为单位)。时钟每秒钟滴答一次(时间自增1);它不跟踪小时或分钟数。软件还应包括一个调度器(scheduler),它将随机安排两个时间,开始每一天。第一个时间是一个人走到第1层,并按下楼层按钮召唤电梯的时间。第二个时间是一个人走到第2层,并按下楼层按钮召唤电梯的时间。类似的每个时间都是随机整数,范围在5~20之间(亦即5,6,7,...,20)。有关如何调度随机时间的问题,将在第3章讨论。假如时钟时间等于这两个时间中较早的那一个,调度器就会创建一个人,令其走到相应的楼层,然后按下楼层按钮。注意,这两个随机安排的时间有可能是一样的;在这种情况下,这个人会同时走到两个楼层,并同时按下两个楼层按钮。楼层按钮变亮,表明它已被按下。注意,楼层按钮按下后变亮是自动发生的,无需编程;按钮被重置后,按钮灯会自动熄灭。电梯每天在第1层关门等待。为省电,电梯只有在需要时才移动。电梯可选择向上和向下移动。

为简化起见,电梯和每个楼层的“载客量”都只有一个人。调度器首先查实一个楼层未被占据,然后创建一个人,让他走进相应的层。假如楼层已被占据,调度器将推迟1秒创建人(使电梯有机会搭载那个人,并清空楼层)。人走进电梯后,调度器会创建下一个随机时间(未来的5~20秒内),届时再让一个人走到楼层,并按下楼层按钮。

电梯抵达一个楼层时,它会重置电梯按钮,并让电梯铃响(已集成到电梯内部)。然后,电梯向楼层发出已抵达的信号。作为响应,楼层会重置楼层按钮,并显示电梯已到达指定楼层的。然后,电梯门开。注意,楼层上对应的门是随电梯门一起自动打开的,无需编程。随后,电梯中的乘客(如果有的话)离开电梯,正在等电梯的人(如果有的话)进入电梯。尽管每个楼层只能容纳一个人,但我们假定每一层都有人等电梯,同时有人离开电梯。

一个人进入电梯后,按下电梯按钮,指示灯会变亮(自动的,无需编程)。另外,等电梯到达另一楼,并重置了电梯按钮后,灯就会熄灭。注意,由于只有两层楼,所以只需要一个电梯按钮;这个按钮的作用仅仅是指示电梯移动到另一层。接着,电梯关门,开始移向另一层。

电梯到达一个楼层后,假如没有人进入电梯,而且另一个楼层上的楼层按钮没有被按下,就关门等待,直到楼层按钮被按下。

为简化起见,假定电梯从抵达一个楼层,在它关门之前,所采取的一切行动都是不花时间的。注意,尽管这些行动不花时间,但仍是按顺序进行的;例如,必须在乘客离开电梯之后,才能关闭电梯门。从一层移向另一层,电梯要花5秒钟。在每一秒钟,模拟程序都要向调度器和电梯提供时间。调度器和电梯将分别利用这个时间,决定在那个特定时间应采取的行动。例如,调度器可以判断是否到时间创建一个人;而电梯在移动过程中,可判断是否到达了目标楼层。

模拟程序应在屏幕上显示消息,描述系统中正在发生的行动。其中包括有人按下了楼层按钮,电梯到达了一个楼层,时钟正在滴答,有人进入电梯……等等。其输出结果如下所示:

```
Enter run time:30
(scheduler schedules next person for floor 1 at time 5)
(scheduler schedules next person for floor 2 at time 17)

* * * ELEVATOR SIMULATION BEGINS * * *

TIME:1
elevator at rest on floor 1

TIME:2
elevator at rest on floor 1

TIME:3
elevator at rest on floor 1

TIME:4
elevator at rest on floor 1

TIME:5
scheduler creates person 1
person 1 steps onto floor 1
person 1 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 20)
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 1 enters elevator from floor 1
person 1 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
```

elevator begins moving up to floor 2(arrives at time 10)

TIME:6

elevator moving up

TIME:7

elevator moving up

TIME:8

elevator moving up

TIME:9

elevator moving up

TIME:10

elevator arrives on floor 2

elevator resets its button

elevator rings its bell

floor 2 resets its button

floor 2 turns on its light

elevator opens its door on floor 2

person 1 exits elevator on floor 2

floor 2 turns off its light

elevator closes its door on floor 2

elevator at rest on floor 2

TIME:11

elevator at rest on floor 2

TIME:12

elevator at rest on floor 2

TIME:13

elevator at rest on floor 2

TIME:14

elevator at rest on floor 2

TIME:15

elevator at rest on floor 2

TIME:16

elevator at rest on floor 2

TIME:17

scheduler creates person 2

person 2 steps onto floor 2

person 2 presses floor button on floor 2

floor 2 button summons elevator

(scheduler schedules next person for floor 2 at time 34)  
elevator resets its button  
elevator rings its bell  
floor 2 resets its button  
floor 2 turns on its light  
elevator opens its door on floor 2  
person 2 enters elevator from floor 2  
person 2 presses elevator button  
elevator button tells elevator to prepare to leave  
floor 2 turns off its light  
elevator closes its door on floor 2  
elevator begins moving down to floor 1(arrives at time 22)

TIME:18  
elevator moving down

TIME:19  
elevator moving down

TIME:20  
scheduler creates person 3  
person 3 steps onto floor 1  
person 3 presses floor button on floor 1  
floor 1 button summons elevator  
(scheduler schedules next person for floor 1 at time 26)  
elevator moving down

TIME:21  
elevator moving down

TIME:22  
elevator arrives on floor 1  
elevator resets its button  
elevator rings its bell  
floor 1 resets its button  
floor 1 turns on its light  
elevator opens its door on floor 1  
person 2 exits elevator on floor 1  
person 3 enters elevator from floor 1  
person 3 presses elevator button  
elevator button tells elevator to prepare to leave  
floor 1 turns off its light  
elevator closes its door on floor 1  
elevator begins moving up to floor 2(arrives at time 27)

TIME:23  
elevator moving up

TIME:24

```

elevator moving up

TIME:25
elevator moving up

TIME:26
scheduler creates person 4
person 4 steps onto floor 1
person 4 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 35)
elevator moving up

TIME:27
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 3 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1(arrives at time 32)

TIME:28
elevator moving down

TIME:29
elevator moving down

TIME:30
elevator moving down

*** ELEVATOR SIMULATION ENDS ***

```

我们的目标(通过第2~7章和第9章的“对象思想”小节)是实现一个能实际运行的模拟程序,在由用户输入的秒数期间,对电梯的运行进行建模。

### 2.22.2 分析和设计系统

在这里以及随后的几个“对象思想”小节中,我们针对这个电梯模拟系统,依次逐步执行面向对象的设计过程。UML设计用于任何OOAD过程——类似的过程较多。其中最流行的方法之一是Rational Software(瑞理软件公司)开发的Rational Unified Process。针对这个案例分析,我们将为你的第一次OOD/UML体验,展示一个简化的设计过程。

设计前,必须先了解模拟的本质。模拟由两部分构成。其一包含了我们想模拟的所有元素。这些元素包括电梯、楼层、按钮、指示灯等等。我们把这一部分叫做世界部分。其二包含对这个世界进行模拟所需的所有元素。这些元素包括时钟和调度器。我们把这一部分



叫做控制部分。设计系统时,务必仔细区分这两部分。

### 2.22.3 用例图

开发者开始新项目时,极少从撰写问题陈述开始(在本节一开始便提供了一个问题陈述的例子)。这样的文档以及其他类似文档通常是面向对象分析(OOA)阶段所产生的结果。在这个阶段,你需要会晤客户以及用户。利用获取的信息,汇编一个系统需求列表。在你和你的伙伴设计系统时,这些需求将为你提供指引。在我们的案例分析中,问题陈述包含了电梯系统的系统需求。对于分析阶段的输出来说,它的意图是清楚地指出系统应如何构建才能完成需要它做的工作。

UML 提供了用例图,以便简化进行需求分析的过程。用例图可针对系统外部客户同系统的用例之间的交互,进行建模分析。每一个用例都代表系统提供给客户的一种不同能力。例如,自动提款机便有几个用例,包括“存款”、“取款”和“转帐”。

图 2.39 展示了电梯系统的用例图。其中的线条画代表的是一名执行者(actor)。执行者包括任何外部实体,比如人、机器人以及其他系统等等,它们都需要使用这个系统。显然,在我们的系统中,惟一的执行者便是想搭乘电梯的人。所以,我们干脆建模了一个名为“Person”的执行者。执行者的“名称”出现在线条画的下方。

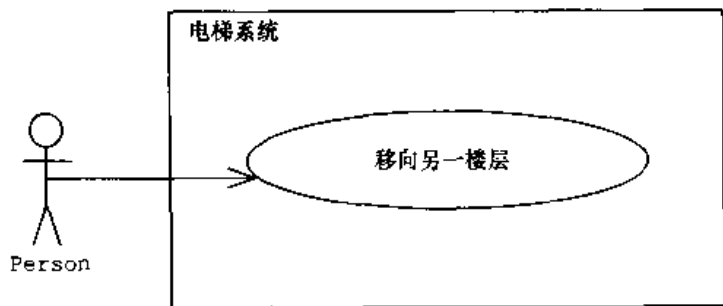


图 2.39 电梯系统的用例图

系统框架(system box)(亦即图中封闭的矩形)包含了系统的用例。注意这个框架被标记为“电梯系统”。这表示该用例模型重点反映了我们想要模拟的系统的行为(即“用电梯搭人”),而非模拟的行为(即“创建人,并安排到达”)。

UML 用椭圆来代表用例。在我们这简单的系统中,执行者只用电梯做一件事情:移向另一楼层。系统只为它的用户提供了一种功能;所以,在我们的电梯系统中,“移向另一楼层”便是惟一的用例。

构建系统时,需依赖用例图保证满足客户的所有需求。我们的案例分析只包含了一种用例。在较大的系统中,系统设计人员为了一直能集中精力满足用户的需求,用例图将成为他们不可或缺的工具。用例图的目标是展现用户同系统之间的各种交互,但不必提供交互的细节。

### 2.22.4 标识系统中的类

在我们的 OOD 过程中,下一步是标识系统中的类。我们最终将用正规方式来描述类,并用 C++ 来实现它们(第 6 章开始将用 C++ 实现电梯模拟程序)。首先审查问题陈述,挑出

其中的所有名词;这些名词极有可能代表了实现电梯模拟程序所需的大多数类(或类的实例)。图 2.40 对这些名词进行了总结。

问题陈述中的名词
company(公司)
building(建筑物)
elevator(电梯)
simulator(模拟程序)
clock(时钟)
time(时间)
scheduler(调度器)
person(人)
floor 1(第1层)
floor button(楼层按钮)
floor 2(第2层)
elevator door(电梯门)
energy(电力)
capacity(载客量)
elevator button(电梯按钮)
elevator bell(电梯铃)
floor's elevator arrival light(楼层的电梯到达指示灯)
person waiting on a floor(在楼层等电梯的人)
elevator's passenger(电梯的乘客)

图 2.40 问题陈述中的名词

我们只希望选择系统中担负重要职责的名词,因而省略了以下名词:

- company(公司);
- simulator(模拟程序);
- time(时间);
- energy(电力);
- capacity(载客量)。

根本不需要将“公司”建模成一个类,因为公司并不属于模拟系统的一部分;公司只是希望我们为电梯建模。“模拟程序”是完整的C++程序,而非一个可以独立的类。“时间”只是时钟的一种属性,不能自成为一个实体。在我们的模拟中,也不会对“电力”进行建模(尽管电力、汽油或石油公司也许会乐于在它们的模拟程序中这样做)。最后,“载客量”只是电梯和楼层的一种属性——本身不能成为一个实体。

为确定系统中真正需要的类,我们将剩余的各个名词划分为不同的类别。事实上,图 2.40 中剩下的所有名词都可从属于以下一个或多个类别:

- building(建筑物);
- elevator(电梯);
- clock(时钟);
- scheduler(调度器);

- person(person waiting on a floor, elevator's passenger)(人(在楼层等电梯的人,电梯的乘客));
- floor(floor 1, floor 2)(楼层(第1层,第2层));
- floor button(楼层按钮);
- elevator button(电梯按钮);
- bell(铃);
- light(灯);
- door(门)。

这些类别都有可能成为最终要在系统中实现的类。注意我们分别为楼层上的按钮和电梯中的按钮建立了一个类。这两类按钮在模拟系统中担负不同的职责——楼层上的按钮召唤电梯,而电梯中的按钮指示电梯开始移向另一层。

现在,根据我们派生出来的类别,为系统中的类建模。根据约定,类名的首字母应大写。假如一个类名同时包含多个单词,那么所有单词都应连写,而且每个的首字母都应大写(如 MultipleWordName)。按这个约定,我们将创建以下几个类: Elevator(电梯), Clock(时钟), Scheduler(调度器), Person(人), Floor(楼层), Door(门), Building(建筑物), FloorButton(楼层按钮), ElevatorButton(电梯按钮), Bell(铃)和 Light(灯)。

### 2.22.5 类图

UML 允许我们通过类图对电梯系统中的“类”及其“关系”进行建模。图 2.41 展示了如何用 UML 表示一个类。其中,我们想要建模的是 Elevator 类。在一个类图中,各类都用矩形来表示。然后,该矩形被分成 3 部分。顶部包含类名。



图 2.41 用 UML 表示一个类

中部包含类的属性;有关属性的详情,参见第 3 章末尾的“对象思想”小节。底部包含类的操作;有关操作的详情,参见第 4 章末尾的“对象思想”小节。

通过关联,便将不同的类联系在一起。图 2.42 展示了 Building、Elevator 和 Floor 类之间的关联关系。注意该图中的矩形并未分成 3 部分。UML 允许用这种方式简化类符号,以便创建更具可读性的类图。

在这个类图中,一条实线将不同的类连接起来,实线代表一个关联。所谓关联,就是类之间的一种关系。实线上的数字表示多重性的值。这个值指出一个类有多少对象参与了这个关联。从上图中,可看出 Floor 类有 2 个对象参与了同 Building 类的一个对象的关联。因此,我们说 Building 类与 Floor 类有一种一对一的关系;还可说, Floor 类与 Building 类有一种二对一的关系。从图中还可看出, Building 类与 Elevator 有一种一对一关系;反之亦然。使用 UML 时,我们可对不同类型的多重性进行建模。图 2.43 总结了不同的多重性类型及其含义。

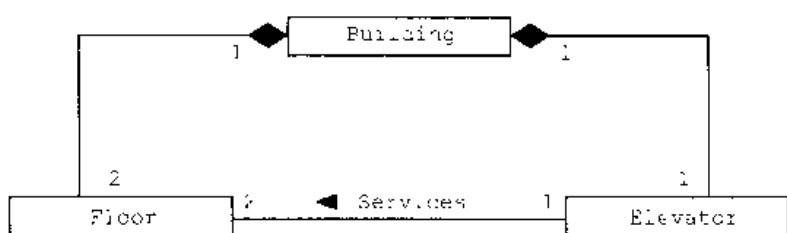


图 2.42 在类图中,表达不同类之间的“关联”

符号	含义
0	无
1	1
$m$	1 个整数值
0..1	0 或 1
$m..n$	至少为 $m$ , 但不大于 $n$
*	任意非负整数
0..*	0 或更多
1..*	1 或更多

图 2.43 多重性类别总结

可为一个关联命名。例如,在连接 Floor 和 Elevator 这两个类的直线上,单词“Services”(服务)指出的便是那个关联的名称——箭头则指示关联方向。在那个图中,这一部分应读作:“Elevator 类的一个对象为 Floor 类的两个对象提供服务”。

在 Building 类的关联线上,实心菱形表明,Building 类同 Floor 和 Elevator 类之间有一种合成关系;或者说,Building 类由 Floor 类和 Elevator 类合成。“合成”暗示着这是一种整体/部分关系。对于关联线末尾有合成符号(实心菱形)的类来说,它便是整体(本例便是 Building);而关联线另一端的类便是部分(如 Floor 和 Elevator)<sup>①</sup>。

图 2.44 展示了电梯系统的完整类图。我们创建的所有类都已建模,并规定了这些类之间的关联(我们将注意在第 9 章采用“继承”这一面向对象的概念,对该类图进行扩展)。

Building 类接近于类图顶部,由 4 个类合成,其中包括 Clock 和 Scheduler。这两个类构成了模拟系统的控制部分<sup>②</sup>。Building 类另外还合成了 Elevator 和 Floor 类(注意 Building 和 Floor 类的“一对二”关系)。

Floor 和 Elevator 类建模于类图底部。Floor 类同时由 Light 类和 FloorButton 类的一个对象构成。Elevator 类则同时由 ElevatorButton、Door 和 Bell 类的一个对象合成。

牵涉到“关联”中的类也可以有自己的角色(roles)。角色有助于澄清两个类之间的关

① 按照 UML 1.3 规范,合成关系中的类将符合 3 种特征:1) 关系中只能有一个类代表“整体”(也就是说,菱形只能放在关联线的一端);2) 合成意味着“整体”的生命期同“整体”相符,而“整体”负责创建和删除它的“部分”;3) 一个“部分”一次只能从属于一个“整体”,但“部分”可以删除,并可同另一个“整体”连接;然后,另一个“整体”将担任这个“部分”的创建和删除。

② Building(建筑物)和 Clock(时钟)/Scheduler(调度器)类之间的合成关系代表了由我们自行决定的一种设计决策。我们认为 Building 同时属于模拟系统的“世界”和“控制”部分。在设计中,我们令建筑物负责模拟系统的运行。

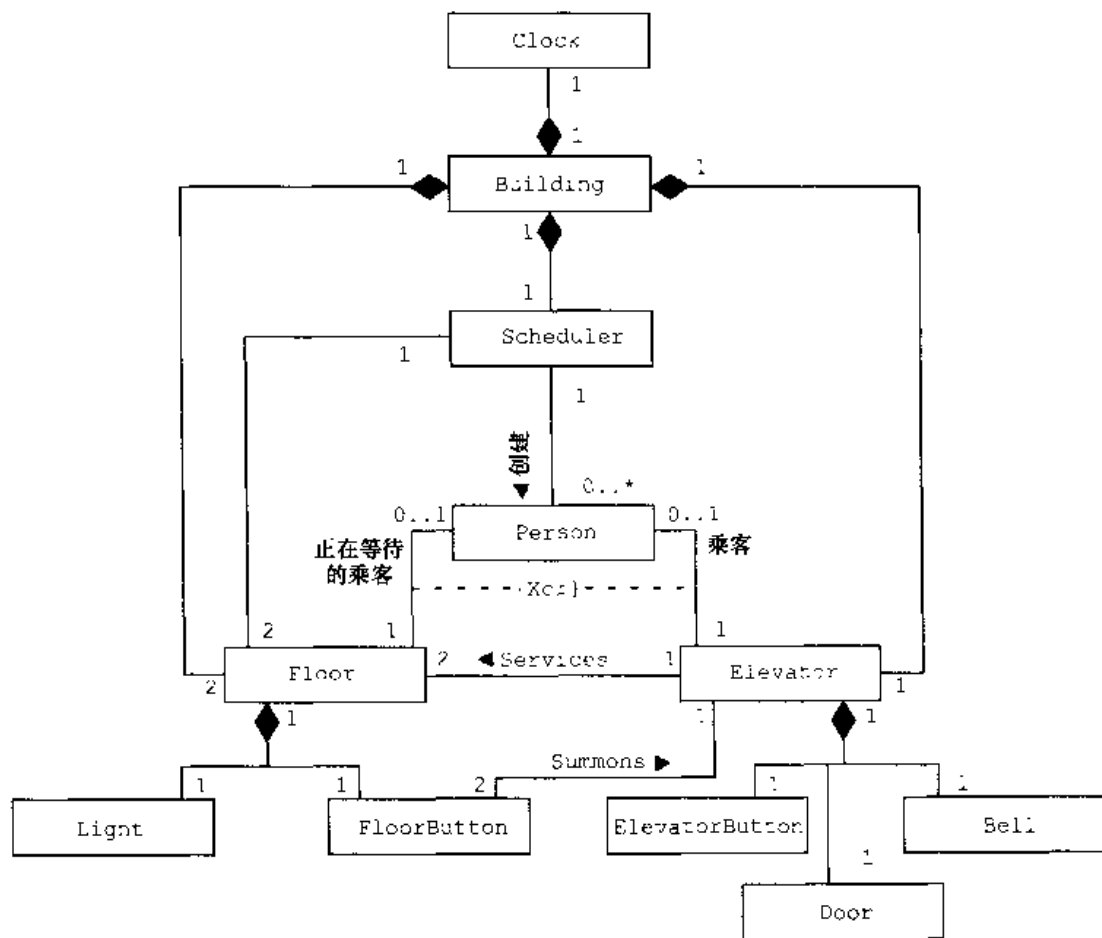


图 2.44 电梯模拟系统的完整类图

系。例如, Person 类在同 Floor 类的关联中, 扮演了“waiting passenger”(正在等候的乘客)的角色(因为人正在等电梯)。Person 类在同 Elevator 类的关联中, 扮演了“passenger”(乘客)的角色。在一个类图中, 类的角色名称可放在关联线的任何一端, 而且可位于类的矩形附近。关联中的每个类都可扮演不同的角色。

Floor 和 Person 类之间的关联指出, Floor 类的一个对象可同 Person 类的 0 个或 1 个对象建立关联。Elevator 类也可同 Person 类的 0 个或 1 个对象建立关联。在这两条关联线之间, 还有一条专门连接起来的虚线, 它指出 Person、Floor 和 Elevator 这 3 个类的关系有一定的限制。这意味着, Person 类的一个对象可参与同 Floor 类的一个对象的关系中, 或参与 Elevator 类的一个对象的关系中, 但不可以同时与两个对象关联。我们用“xor”字样标识这种关系(xor 代表“异或”, 或“exclusive or”), 注意“xor”要放在一对花括号中<sup>①</sup>。Scheduler 和 Person 类之间的关联指出, Scheduler 类的一个对象将创建 Person 类的 0 个或多个对象。

## 2.22.6 对象图

UML 还定义了对象图, 它同类图相似, 只是它建模的是对象和链接(链接是对象之间的

<sup>①</sup> UML 图示中的限制可用所谓的“对象限制语言”(OCL)编写。利用 OCL, 建模人员可用明确定义的方式表达系统中的各种限制。欲知详情, 请访问 [www-4.ibm.com/software/ad/library/standards/ocl.html](http://www-4.ibm.com/software/ad/library/standards/ocl.html)。

关系)。和类图相同,对象图也对系统的结构进行建模。对象图展示了系统运行时的一个结构“快照”,即提供了在某个特定的时刻,参与系统运作的各个对象的信息。

图 2.45 展示了建筑物中没有任何人时,我们所取得的系统模型快照(此时,系统中不存在任何 Person 类的对象)。对象名通常写成这种形式:objectName : ClassName。在对象名中,第一个单词的首字母不大写,后面的单词则需要。对象图中的所有对象名都要加下划线。在类图中,我们省略了一些对象的对象名(比如 FloorButton 类的对象)。在大型系统中,模型中通常会用到许多对象名。这极易产生混乱的、难以辩读的图示。假如一个特定对象的名称未知,或者没必要包括那个名称(换言之,我们关心的只是对象类型),则可考虑删除对象名,只显示冒号和类名。

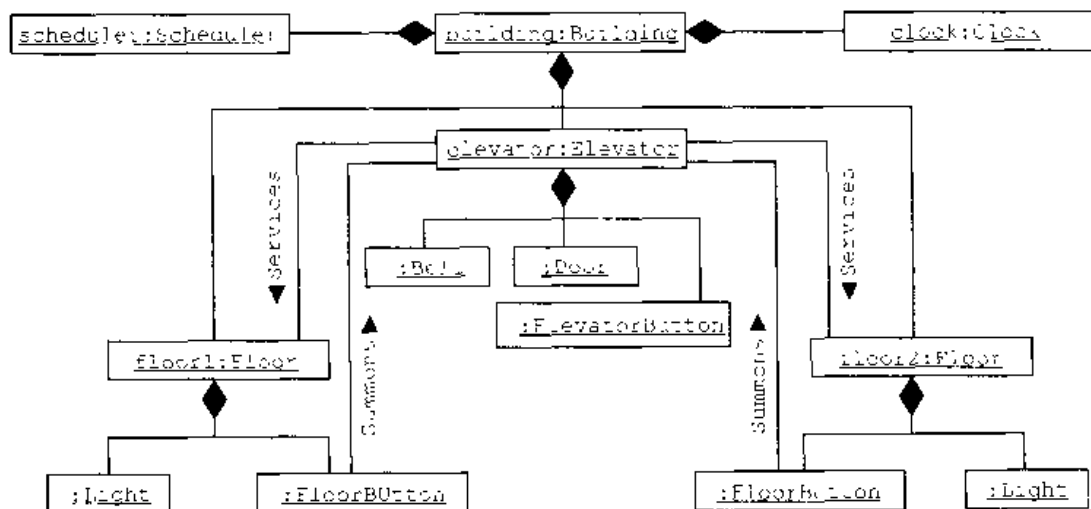


图 2.45 空建筑物的对象图

现在,我们已标识好了这个系统中的类(尽管在设计过程的后续阶段中,还有可能发现其他类)。此外还探讨了系统的用例。第3章末尾的“对象思想”小节,还将利用这里介绍的知识探讨系统随着时间变化而产生的变化。在不断地拓展现有知识的同时,还会发现一些新的信息,它们有助于我们更深入探讨现有的类。

#### 说明:

(1) 第3章将介绍如何实现随机性,如何生成随机数。利用随机数生成,可有效模拟一些随机的过程,比如抛硬币和掷骰子等等。另外,它也能帮助你模拟在随机的时间创建人,并开始使用电梯。

(2) 由于现实世界如此“面向对象”,所以该项目将非常容易理解——即使你还未正式学过面向对象的技术。

#### 问题:

(1) 如何判断电梯是否能处理预期的承载量。

(2) 为什么实现1幢3层建筑物(或者更高)会复杂得多?

(3) 大型建筑一般都有多部电梯。在第6章,你会发现一旦创建了一个电梯对象,就可根据需要很容易地创建更多的电梯对象。在大型建筑物中,假如同时有多部电梯,每一部都

可在同一楼层接送多名乘客,估计会出现哪些急待解决的问题(或者情况)?

(4) 为简化起见,我们假定每部电梯和每个楼层都只能承载一名乘客。假如增大承载量,会出现哪些新问题?

## 2.23 小结

- 解决问题时,具体过程即为“要采取的行动”;采取的顺序即为“算法”。
- 在计算机程序中,由程序规定语句执行顺序即为“程序控制”。
- 伪代码帮助程序员先“思考”程序,再用类似C++的编程语言编写程序。
- 声明是传给编译器的消息,告诉它变量的名称和属性,并要求它为变量预留空间。
- 选择结构用于在各种候选的行动中做出选择。
- if 选择结构只有在条件为 true 时,才会采取指定的行动。
- if/else 选择结构指定在条件为 true 时,采取行动;在条件为 false 时,采取另一项行动。
- 假如打算执行的是多条语句,但当时的环境决定了只能执行一条语句,这些语句就必须用花括号封闭起来,从而构成一条复合语句。复合语句可放在原本只能放置一条语句的任何地方。
- 空语句指出不采取任何行动,做法是在本该放置语句的地方,使用分号(;)。
- 重复结构表明,在条件保持 true 的前提下,将反复地采取某种行动。
- while 反复结构的格式如下:
 

```
while (conditions)
    statement
```
- 包含了小数部分的值叫做浮点数,用 float 或 double 这两种数据类型表示。
- 一元强制类型转换操作符 `static_cast < double >()` 可创建其操作数的一个临时浮点副本。
- C++ 提供了算术赋值操作符 `+=`、`-=`、`*=`、`/=` 和 `%=`,有助于对特定的通用类型表达式进行简化。
- C++ 提供了自增(`++`)和自减(`--`)操作符,可使一个变量自增或自减 1。假如操作符放在变量名之前,会先使变量自增或自减 1,再开始在表达式中使用变量;假如操作符放在变量名之后,会先在表达式中使用变量,再使变量自增或自减 1。
- 循环是一组指令,计算机会重复执行它们,直到符合某种中止条件。两种形式的重复是由计数器控制的重复,以及由标记控制的重复。
- 循环计数器用于统计一组指令的重复次数。每执行一遍这组指令,它都会自增或自减(通常是 1)。
- 假如提前不知道准确的重复次数,而且循环中包含了每次循环时都要求获取数据的语句,那么通常用标记来控制重复。将全部有效的数据项提供给程序后,便可输入标记。标记应当同有效的数据项有所区别。
- for 重复结构处理“由计数器控制的重复”的所有细节。for 结构的常规格式如下:
 

```
for ( initialization; loopContinuationTest; increment )
```

```
statement
```

其中,“初始化”表达式用于对循环的控制变量进行初始化;“循环继续检测”是检测循环是否应继续的一个条件;而“自增”用于使控制变量自增。

- do/while 重复结构在循环结束时才检测循环继续条件,所以循环主体至少会执行一遍。do/while 结构的格式为:

```
do
    statement
while (condition);
```

- break 语句在某个重复结构(for、while 和 do/while)中执行时,会导致立即从结构中退出。
- continue 语句在某个重复结构(for、while 和 do/while)中执行时,会导致跳过结构主体中剩余的所有语句,直接开始下一次循环。
- switch 语句处理的是一系列决定;其中,会针对它能假定的值检测特定的变量或表达式,并根据结果采取不同的行动。在大多数程序中,都有必要为每个 case 都包括 break 语句。几个 case 也可执行相同的语句,做法是将所有 case 标签都列于语句之前。switch 结构只能测试常量整数表达式。不必将含有多条语句的 case 封闭在花括号中。
- 在 Unix 系统和其他系统中,文件结束(EOF)的输入标记是在一行中单独按以下组合键

```
<Ctrl - d>
```

在 VMS 或 DOS 下,文件结束的输入可采用以下组合键

```
<Ctrl - z>
```

然后可能还要按一次回车键。

- 可用逻辑操作符合并不同的条件,从而构成复杂的条件。逻辑操作符包括 &&(逻辑 AND),|| (逻辑 OR)以及!(逻辑 NOT)。
- 非零值的默认含义是 true;0(零)的默认含义是 false。

## 本章术语

! operator ! 操作符

&& operator && 操作符

|| operator || 操作符

++ operator ++ 操作符

-- operator -- 操作符

?: operator ?: 操作符

action/decision model 行动/决策模型

algorithm 算法

arithmetic assignment operators

算术赋值操作符: +=, -=, /= 和 %=

ASCII character set ASCII 字符集

block 代码块

body of a loop 循环主体/循环体

case label case 标签

cast operator 强制类型转换操作符

cin.get() function cin.get() 函数

compound statement 复合语句

conditional operator(?:) 条件操作符(?:)

control structure 控制结构

counter-controlled repetition 计数器控制重复

decrement operator(--) 自减操作符(--)

default case in switch switch 中的 default 条件

definite repetition 无限循环

definition 定义

delay loop 延迟循环

do/while repetition structure do/while 重复结构



double-selection structure 双选结构  
 empty statement(;) 空语句(;)   
 fatal error 致命错误  
 field width 域宽  
 fixed-point format 定点格式  
 for repetition structure for 重复结构  
 garbage value 垃圾值  
 if selection structure if 选择结构  
 if/else selection structure if/else 选择结构  
 increment operator(++) 自增操作符(++)  
 indefinite repetition 不确定重复  
 infinite loop 无限循环  
 initialization 初始化  
 integer division 整除  
 keyword 关键字  
 logic error 逻辑错误  
 logical AND(&&) 逻辑 AND(&&)  
 logical negation(!) 逻辑非(!)  
 logical operators 逻辑操作符  
 logical OR(||) 逻辑 OR(||)  
 loop counter 循环计数器  
 loop-continuation condition 循环继续条件  
 looping 循环  
 left value(lvalue) 左值(lvalue)  
 multiple-selection structure 多选结构  
 nested control structures 嵌套控制结构  
 nonfatal error 非致命错误  
 off-by-one error 相差1错误  
 parameterized stream manipulator 参数化流操纵元  
 postdecrement operator 后自减操作符  
 postincrement operator 后自增操作符

## “对象思想”术语

actor 执行者(actor)  
 association 关联  
 association name 关联名称  
 class diagram 类图  
 composition 合成  
 constraint 限制  
 controller portion of a simulation  
   一个模拟系统的控制部分  
 identify the classes in a system 标识系统中的类  
 link 链接  
 multiplicity 多重性

pow function pow 函数  
 predecrement operator 前自减操作符  
 preincrement operator 前自增操作符  
 pseudocode 伪代码  
 repetition 重复  
 repetition structures 重复结构  
 right value(rvalue) 右值(rvalue)  
 selection 选择  
 sentinel value 标记  
 sequential execution 顺序结构  
 setiosflags stream manipulator setiosflags 流操纵元  
 setprecision stream manipulator  
   setprecision 流操纵元  
 setw stream manipulator setw 流操纵元  
 single-entry/single-exit control structures  
   单人/单出控制结构  
 single-selection structure 单选结构  
 stacked control structures 控制结构堆叠  
 static\_cast <type>() static\_cast <类型>()  
 structured programming 结构化编程  
 switch selection structure switch 选择结构  
 syntax error 语法错误  
 ternary operator 三元操作符  
 top-down, stepwise refinement 自上而下求精法  
 transfer of control 转交控制权  
 unary operator 一元操作符  
 undefined value 未定义值  
 while repetition structure while 重复结构  
 whitespace characters 空白字符  
 || operator ||操作符

Object Constraint Language (OCL) 对象限制语言  
 (OCL)

object diagram 对象图  
 object-oriented analysis 面向对象分析(OOA)  
 object-oriented analysis and design(OOAD)  
   面向对象分析和设计(OOAD)  
 object-oriented design 面向对象设计(OOD)  
 one-to-one relationship 一对一关系  
 one-to-two relationship 一对二关系  
 rectangle symbol in UML class and object diagram  
   UML类图中的矩形符号

role 角色	system requirements 系统需求
software simulator 模拟软件	two-to-one relationship 二对一关系
solid diamond symbol in UML class and object diagram UML 类和对象图中的实心菱形符号	use case 用例
solid line symbol in UML class and object diagram UML 类和对象图中的实线符号	use case diagram 用例图
static structure of a system 静态系统结构	what vs. how 是什么和怎么做
system box 系统框架	world portion of a simulation 模拟系统的世界部分
	xor 异或

## 常见编程错误

- 2.1 将关键字用作标识符是语法错误。
- 2.2 如忘记用一个或两个花括号为复合语句定界,会导致语法或逻辑错误。
- 2.3 在 if 结构中,如在条件之后放置分号,那么假如是单选 if 结构,会造成逻辑错误;假如是双选 if 结构,则会造成语法错误(假如 if 部分包含了实际的主体语句)。
- 2.4 在 while 结构的主体,假如不提供最终使 while 条件变成 false 的行动,会造成所谓的“无限循环”错误。在这种情况下,重复结构永远不会中止,会无休止地循环下去。
- 2.5 如将关键词 while 误拼成 While,会导致语法错误(记住,C++ 是一种要严格区分大小写的语言)。C++ 的所有保留关键词,比如 while,if 和 else 等,都只能采用小写字母。
- 2.6 假如不初始化 counter 或 total,会造成不正确的程序结果。这属于逻辑错误。
- 2.7 在计数器控制的循环中,由于循环计数器(通过循环每次都累加 1 的时候)比它最后一个合法值大 1(在从 1 计数到 10 之后,变成 11),所以在循环之后的某次计算中,再使用计数器的值会造成“相差 1”错误。
- 2.8 如挑选的标记同时也是合法的输入数据,就会造成逻辑错误。
- 2.9 被零除会造成严重错误。
- 2.10 使用浮点数时,不可假定它们肯定能精确地表示,否则会导致不确切的结果。在大多数计算机上,浮点数都是约数。
- 2.11 对比较复杂的变量名应用自增或自减操作符——例如 ++(x + 1)——会出现语法错误。
- 2.12 由于浮点值只是近似值,所以用浮点变量对计数循环进行控制,会出现不准确的计数器值,并会出现对中止条件的不准确检测。
- 2.13 假如在 while 或 for 结构的条件中使用了不正确的关系操作符,或者使用了不正确的循环计数器终值,便可能导致值相差 1 错误。
- 2.14 假如一个 for 结构的控制变量最早是在 for 结构头部的初始化小节定义的,那么在结构的主体之后,再使用控制变量便会导致语法错误。
- 2.15 在 for 结构的头部语句,假如误将两个分号写成了逗号,会造成语法错误。
- 2.16 如果在 for 结构的头部语句的右括号之后,紧接着便加分号,会导致 for 结构的主体变成空语句。这通常会造成逻辑错误。
- 2.17 在循环的循环继续条件中,假如未能正确地使用关系操作符,以便进行倒数(比如在循环中不正确地使用  $i \leq 1$  倒数至 1),通常会造成逻辑错误,使程序运行时出现不正确的结果。

- 2.18 在使用了数学库函数的程序中忘了包括 `<cmath>`, 属于语法错误。
- 2.19 忘了在 `switch` 结构中放置必要的 `break` 语句, 会造成逻辑错误。
- 2.20 如果在“case”字样同 `switch` 结构中检测的整数值之间遗漏了空格, 可能导致逻辑错误。例如, 假如写成 `case3:`, 而不是 `case 3:`, 结果是创建一个无用的标签(详情参见第 18 章)。也就是说, 假如 `switch` 的控制表达式的值是 3, `switch` 结构就无法采取恰当的行动。
- 2.21 每次读取字符时, 假如不对输入的换行和其他“空白”字符进行处理, 会导致逻辑错误。
- 2.22 如在 `switch` 结构中提供相同的标签, 会造成语法错误。
- 2.23 在 `while`、`for` 或 `do/while` 结构中, 假如循环继续条件永远变不成 `false`, 就会产生无限循环。为避免这个问题, 一定要在循环头或主体的某个地方更改条件值, 使条件最终能变成 `false`。
- 2.24 尽管从数学上看, 像  $3 < x < 7$  这样的表达式是完全正确的, 但在 C++ 里是绝对不允许的。应该写成: `(3 < x && x < 7)`。
- 2.25 在使用了 `&&` 操作符的表达式中, 有可能一个条件——我们把这种条件称为“依赖条件”——要求在另一个条件为 `true` 的前提下, 才有必要对其进行求值。在这种情况下, 依赖条件应放在另一个条件之后, 否则会出错。
- 2.26 无论用 `==` 赋值, 还是用 `=` 判断是否相等, 都属于典型的逻辑错误, 但不会被视为语法错误——歧义因此而产生。

## 良好编程习惯

- 2.1 在程序中合理进行缩进(缩排)处理, 可显著增强程序的可读性——建议将每个缩进单位设为 1/4 英寸或 3 个空格字符。
- 2.2 在程序设计阶段, 通常先用伪代码来“思考”程序, 再将伪代码程序转换成真正的 C++ 程序。
- 2.3 `if/else` 结构的两个主体语句都应缩进。
- 2.4 如同时有数级缩进, 那么每级缩进都在上一级缩进的基础上增加相同数量的空格。
- 2.5 始终记得在 `if/else` 结构(或其他任何控制结构)中放置花括号, 这样有助于避免它们不慎被遗忘, 特别是在以后为 `if` 或 `else` 从句添加语句时。
- 2.6 有的程序员在花括号内键入单独的语句之前, 习惯于先输好复合语句的起始和结束花括号。这是一个很好的习惯, 有助于避免不慎漏掉一个或两个花括号。
- 2.7 无论如何都要初始化计数器和总和。
- 2.8 单独用一行声明每个变量。
- 2.9 进行除法运算时, 假如除数可能为零, 请务必明确检测这一条件, 并在程序中预先采取防范措施(比如打印一条出错提示信息等等), 不要让潜在的问题引发严重错误!
- 2.10 每次需要键盘输入时都提醒用户。提醒时, 应指出输入所采取的形式, 以及任何特殊值(比如用户应输入哪个标记来中止循环)。
- 2.11 在标记控制的循环中, 请在输入数据提示中, 明确告诉用户哪一个是标记。
- 2.12 不要试图对比两个浮点数是否相等。相反, 应测试两个浮点数的差值的绝对值是否小

于指定的小值。

- 2.13 声明变量时便对将其初始化,有助于程序员避免以后忘记初始化数据的问题。
- 2.14 一元操作符应紧挨操作数,中间不含任何空格。
- 2.15 用整数值来控制计数循环。
- 2.16 每个控制结构主体中的语句都进行缩进处理。
- 2.17 在每个控制结构前后都留一个空行,将其同程序的其余部分区分开。
- 2.18 嵌套级别过多,会导致程序难于理解。通常应把嵌套控制在3级以内。
- 2.19 控制结构上下的垂直间距,以及在控制结构头部对控制结构主体的缩进,可为程序员营造一种二维外观,从而极大增强可读性。
- 2.20 在 while 或 for 结构的条件中使用终值,并使用关系操作符  $\leq$ ,有助于避免产生值相差 1 错误。例如,对用于打印从 1 到 10 的循环来说,循环继续条件应是 `counter  $\leq$  10`,而不应是 `counter < 10`(后者会产生值相差 1 错误),也不应是 `counter < 11`(尽管仍然是正确的)。许多程序员仍然喜欢所谓的“零基计数”。也就是说,为通过一个循环计数 10 次,先将 `counter` 初始化为零,再将循环继续检测条件设为 `counter < 10`。
- 2.21 在 for 结构的初始化及自增部分,应尽量只使用与控制变量有关的表达式。如还需对其他变量进行处理,不在循环之前进行(前提是它们只执行一次,比如初始化语句),就在循环主体内进行(前提是每次重复都要执行,比如自增或自减语句)。
- 2.22 尽管可在 for 循环主体中更改控制变量的值,但尽量避免这样做,因为可能导致不易察觉的逻辑错误。
- 2.23 尽管 for 之前的语句以及 for 之内的语句经常都可合并到 for 的头部,但尽量避免这样做,因为这会降低程序的可读性。
- 2.24 尽可能将任何控制结构的头部限制在一行之内。
- 2.25 不要用 float 或 double 类型的变量来执行财务计算。不精确的浮点数会造成错误,得到不正确的金额。在练习中,我们探讨了用整数执行金融计算的方法。注意:可选择由第三方厂商提供的 C++ 类库,它们能正确地执行金融计算。
- 2.26 无论如何都在 switch 语句中提供一个 default 条件。在无 default 条件的 switch 语句中,那些没有明确进行检测的情况会被忽略。如包括 default 条件,会使程序员关注对例外情况的需求。某些情况下,不需要进行 default 处理。尽管 switch 结构中的 case 从句和 default 从句可按任意顺序排列,但作为一个良好的习惯,应将 default 从句放在最后。
- 2.27 在 switch 结构中,假如 default 从句列于最后,则不需要为它使用 break 语句。有的程序员包括这个 break 的原因是为了更有条理,以及与其他 case 对应。
- 2.28 有的程序员习惯在 do/while 结构中包含花括号——即使花括号并无实际用途。这样做有助于区分 while 结构和只包含了一条语句的 do/while 结构。
- 2.29 有的程序员觉得 break 和 continue 违背了结构化编程准则。由于这些语句的效果可通过后文介绍的结构化编程技术实现,所以他们不使用 break 和 continue。

#### 性能提示

- 2.1 与罗列大量单选 if 结构相比,只用一个 if/else 结构的速度会快许多。这是由于在最后一

种情况下,只要碰到满足条件的第一个表达式,整个结构便会中止并退出,不必遍历所有表达式。

- 2.2 在嵌套的 if/else 结构中,应首先检测最有可能为 true 的条件。这样一来,程序便可早早地退出 if/else 结构。与先检测不大容易成立的条件相比,这样的设计可显著加快执行速度。
- 2.3 如使用了“简写”的赋值操作符,程序员就可以更快地编写程序,编译器也能更快地编译程序。在使用了“简写”赋值操作符的前提下,有的编译器可生成最终能更快运行的代码。
- 2.4 本书的许多“性能提示”只能稍微提升性能,所以有的读者可能对此不以为然。但是,在需要重复多次的循环中,假如每一次都能“稍微”提升性能,最终带来的性能提升仍然是可观的。
- 2.5 不要在循环中放置值不会发生改变的表达式——但是,即便如此,如今许多高级的优化编译器都会在其生成的机器代码中,自动将这样的表达式排除在循环之外。
- 2.6 许多编译器都包含了优化功能,可改进你编写的代码,但最好还要从一开始便养成一次性写好代码的习惯。
- 2.7 在性能要求较高的环境中(内存宝贵,或要求加快执行速度),请尽可能地使用长度较短的整数。
- 2.8 如使用长度较短的整数,但机器负责处理它们的指令不如处理自然长度的整数有效(例如,非要为它们加上符号扩展),反而会使程序运行速度减慢。
- 2.9 如运用得当,break 和 continue 语句的速度要比后文介绍的结构化编程技术快。
- 2.10 在使用了 && 操作符的表达式中,假如不同的条件是相互独立的,那么将最有可能成为 false 的条件放在最左边。在使用了 || 操作符的表达式中,应把最有可能成为 true 的条件放在最左边。这样可缩短程序的执行时间。

### 可移植性提示

- 2.1 在 C++ 标准中,在一个 for 结构的初始化小节声明的控制变量的作用域不同于老版本 C++ 编译器所声明的作用域。在兼容于 C++ 标准的编译器上,如果对以前采用老版本 C++ 编译器创建的 C++ 代码进行编译,可能会出现中断情况。有两种保守型的编程策略可用于防止此类问题。其一是在每个 for 结构中都用不同的名称定义控制变量;其二是,如果你希望在几个 for 结构中为控制变量使用相同的名称,那么在外部的并且在第一个 for 循环之前定义控制变量。
- 2.2 用于输入“文件结束”(EOF)的组合键与具体的系统有关。
- 2.3 如检测符号常量 EOF,而不是检测 -1,会使程序更易移植。ANSI 标准规定,EOF 取一个负的整数值(但不一定是 -1)。因此,EOF 在不同的系统上可能取不同的值。
- 2.4 由于 int 的长度在不同系统之间是不同的,所以如果要处理的整数超过了 -32 768 ~ 32 767 这一范围,或者希望能在多种不同的计算机系统上运行程序,就可使用 long 整数。
- 2.5 为兼容于 C++ 标准的早期版本兼容,“true”(真)这个 bool 值也可表示成任何非零的值:

而“false”(假)这个 bool 值也可表示成值 0。

## 软件工程知识

- 2.1 无论过去还是将来,我们的所有C++程序都可基于前述7类控制结构(顺序,if,if/else,switch,while,do/while 和 for)进行构建。不同控制结构只需控制结构堆叠和嵌套两种方法即可合并到一起。
- 2.2 程序内凡是可以放置单条语句的地方,都可放置复合语句。
- 2.3 尽管前文提到说:“凡是可以放置单条语句的地方,都可放置复合语句”,但也不排斥另一种情况——不放置任何语句。也就是说,在这些地方,我们可以放置一条空语句。空语句的意思是指:在本该是语句的地方,仅用一个分号(;)来充数,而不采取任何行动。
- 2.4 每次求精(包括最顶部的整体表述)其实都是一个完整的算法规范;只是细化程度有所区别。
- 2.5 许多程序都可在逻辑上分成3个阶段:初始化阶段对变量进行初始化;处理阶段输入数据值,并相应地更改程序变量;结束阶段则计算并打印出最终结果。
- 2.6 只要伪代码算法提供了足够多的细节,利用这些细节可将伪代码轻松转换成C++程序,便可考虑停止“自上而下求精”。之后,根据伪代码即可轻松写出C++程序。
- 2.7 经验表明,用计算机解决现实世界的问题时,最困难的部分便是设计出一个合理的算法。一旦拟出正确的算法,即可轻松将其转换成实际的C++程序!
- 2.8 许多高手在写程序时,根本没用过类似“伪代码”的辅助开发工具。这些程序员认为自己的终极目标便是用计算机解决实际问题。编写伪代码会耽搁时间,影响开发进度。不过要提醒大家的是:一方面,你可能实际并非实际意义上的“高手”;另一方面,不编写伪代码也许足以应付简单而熟悉的问题,但对一些大型的、复杂的项目,恐怕困难重重。
- 2.9 有时,可在for头部之后,紧跟一个分号。其目的是创建所谓的延迟循环。特意带有一个空主体的for循环仍会正常循环指定的次数,只是除计数之外,无任何作用。例如,你可利用一个延迟循环放慢程序的执行速度,确保屏幕输出不至于过快,避免无法阅读。
- 2.10 在实现高质量的软件工程和获得最佳的性能之间,总难取得平衡。往往会顾此失彼。

## 测试和调试提示

- 2.1 程序员习惯于写一些类似  $x == 7$  的条件表达式,把变量名放到左边,把常量放到右边。但是,一种更好的做法却是逆转这个顺序,把常量放到左边,把变量名放到右边(比如  $7 == x$ )。这样一来,一旦不慎将  $==$  写成了  $=$ ,编译器立刻就报告错误。以  $7 = x$  为例,编译器看到这样的写法,便会正确判断出它是一个语法错误,因为在赋值语句中,不可以将一个变量赋给一个常数,不可以将常数放在赋值操作符的左边。这样至少可以防止运行时逻辑错误对程序的干扰。
- 2.2 用文本编辑器查找程序中的所有  $=$ ,查实各处使用的操作符是否正确。

## 自测题

自测题 2.1 ~ 2.10 与 2.1 ~ 2.12 节中的内容对应。自测题 2.11 ~ 2.13 与 2.13 ~ 2.21

节中的内容对应。

### 2.1 填空题:

- a) 所有程序都可用 3 类控制结构写成,它们是:\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- b) \_\_\_\_\_选择结构会在条件为 true 时采取一项行动,条件为 false 时采取另一项行动。
- c) 一系列指令重复执行指定的次数,这称为\_\_\_\_\_重复。
- d) 如事先不知道一系列语句的重复次数,就必须用一个\_\_\_\_\_值来中断重复。

### 2.2 分别写 4 条不同的 C++ 语句,每条语句都能在整数变量 x 上加 1。

### 2.3 针对以下任务,编写 C++ 语句:

- a) 把 x 和 y 的和赋给 z。计算完成后,让 x 的值自增 1。
- b) 测试 count 变量的值是否大于 10。如答案是肯定的,则打印一条消息:“Count is greater than 10”。
- c) 让 x 变量的值减 1,再从 total 变量值中减去 x,结果仍保存在 total 中。
- d) 计算变量 q 除以变量 divisor 后的余数,把结果赋还给 q。请列出这条语句的两种不同的写法。

### 2.4 针对以下任务,分别编写 C++ 语句:

- a) 将变量 sum 和 x 声明成类型 int。
- b) 将变量 x 初始化为 1。
- c) 将变量 sum 初始化为 0。
- d) 将变量 x 同变量 sum 相加,把结果赋给变量 sum。
- e) 打印“The sum is: ”,后接变量 sum 的值。

### 2.5 将自测题 2.4 写的语句合并成一个程序,计算和打印 1~10 的所有整数的和。用 while 结构循环执行计算和自增语句。x 的值变成 11 时,中止循环。

### 2.6 判断计算执行后每个变量的值。假定每条语句开始执行时,所有变量都等于整数值 5。

- a) `product *= x++;`
- b) `quotient /= ++x;`

### 2.7 针对以下任务,编写 C++ 语句:

- a) 用 cin 和 >> 输入整型变量 x。
- b) 用 cin 和 >> 输入整型变量 y。
- c) 将整型变量 i 初始化为 1。
- d) 将整型变量 power 初始化为 1。
- e) 令变量 power 和 x 相乘,把结果赋给 power。
- f) 令 y 变量自增 1。
- g) 测试 y,判断它是否小于或等于 x。
- h) 用 cout 和 << 输出整型变量 power。

### 2.8 编写一个 C++ 程序,使其利用自测题 2.7 中的语句计算 x 的 y 次方。程序应用 while 重复控制结构。

### 2.9 指出下列语句中的错误,并说明如何改正:

```

a) while( c <= 5 ) {
    product *= c;
    ++c;
}
b) cin << value;
c) if ( gender == 1 )
    cout << "Woman." << endl;
else;
    cout << "Man" << endl;

```

2.10 指出下列 while 重复结构中的错误。

```

while ( z >= 0 )
    sum += z;

```

2.11 判断正误。如果有错,请说明原因。

- default 条件是 switch 选择结构必需的。
- 必须在 switch 选择结构的 default 条件中使用 break 语句,才能正确退出结构。
- 假如  $x > y$  这个表达式为 true,或者  $a < b$  这个表达式为 true,那么  $(x > y \&\& a < b)$  这个表达式肯定为 true。
- 对包含了 || 操作符的表达式来说,假如它的两个操作数之一或者全部为 true,该表达式就一定为 true。

2.12 针对以下操作,编写一条或多条 C++ 语句:

- 用 for 结构求 1~99 之间的所有奇数的和。假定整型变量 sum 和 count 已经声明。
- 用 15 个字节的域宽打印数值 333.546 372,精度分别取 1,2,3。在同一行打印所有数字。每个数字在其字段中左对齐。打印出来的 3 个值分别是什么?
- 使用 pow 函数计算 2.5 的 3 次方。用精度 2、域宽 10 打印结果。打印的结果是什么?
- 用一个 while 循环和计数器变量 x 打印 1~20 的所有整数。假定变量 x 已经声明,但未初始化。每行只打印 5 个整数。提示:使用  $x \% 5$  这一计算式。当它的结果为 0 时,便打印一个换行符,否则打印一个制表符。
- 换用一个 for 结构,重做自测题 2.12(d) 部分。

2.13 指出以下代码中的错误,并说明如何改正。

```

a) x = 1;
   while ( x <= 10 ) {
       x++;
       |
   }
b) for ( y = .1; y != 1.0; y += .1 )
    cout << y << endl;
c) switch ( n ) {
    case 1:
        cout << "The number is 1" << endl;
    case 2:
        cout << "The number is 2" << endl;
        break;

```



```

    default:
        cout << "The number is not 1 or 2" << endl;
        break;
    }

```

d) 以下代码用于打印 1 ~ 10 之间的值。

```

n = 1;
while ( n < 10 )
    cout << n++ << endl;

```

### 自测题答案

2.1 a) 顺序, 选择和重复 b) if/else c) 由计数器控制的, 或“确定” d) 标记、信号、旗帜或假

2.2

```

x = x + 1;
x += 1;
++x;
x++;

```

2.3 a) `z = x++ + y;`  
 b) `if ( count > 10 )`  
     `count << "Count is greater than 10" << endl;`  
 c) `total -= --x;`  
 d) `q %= divisor;`  
     `q = q % divisor;`

2.4 答案如下:

a) `int sum, x;`  
 b) `x = 1;`  
 c) `sum = 0;`  
 d) `sum += x;` 或者 `sum = sum + x;`  
 e) `cout << "The sum is: " << sum << endl;`

2.5 答案如下:

```

1 //Calculate the sum of the integers from 1 to 10
2 #include <iostream>
3
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int sum, x;
10    x = 1;
11    sum = 0;
12    while( x <= 10 ) {
13        sum += x;
14        ++x;

```

```

15     ;
16     cout << "The sum is: " << sum << endl;
17     return 0;
18

```

## 2.6 答案如下:

- a) product = 25, x = 6;
- b) quotient = 0, x = 6;

## 2.7 答案如下:

- a) cin >> x;
- b) cin >> y;
- c) i = 1;
- d) power = 1
- e) power \*= x; 或者 power = power \* x;
- f) i++;
- g) if ( i < y)
- h) cout << power << endl;

## 2.8 答案如下:

```

1 //raise x to the y power
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10     int x,y,i,power;
11
12     i=1;
13     power = 1;
14
15     cout << " Enter base as an integer: ";
16     cin >> x;
17
18     cout << "Enter base as an integer:";
19     cin >> y;
20
21     while (i <= y) {
22         power *= x;
23         ++i;
24     }
25
26     cout << power << endl;
27     return 0;
28 }

```

## 2.9 答案如下:

- a) 错误:用于结束 while 主体的右花括号使用不当。  
改正:将结束右花括号添加到 ++c; 语句的末尾。
- b) 错误:使用流插入,而不是流读取。  
改正:把 << 改为 >>。
- c) 错误:else 后的分号造成了逻辑错误。第二个输出语句始终会执行。  
改正:删除 else 之后的分号。

2.10 变量 z 的值在 while 结构中永远不会改变。所以,假如循环继续条件( $z \geq 0$ )为 true,就会产生无限循环。为避免无限循环,z 必须自减,使其最终能小于 0。

## 2.11 答案如下:

- a) 错误:default 条件是可选的。假如无需默认行动,则不必添加 default 条件。
- b) 错误:break 语句用于退出 switch 结构。假如有 default 条件作为最后一个 case,则不需要 break 语句。
- c) 错误:使用 && 操作符时,两个关系表达式都为 true 时,整个表达式才能为 true。
- d) 正确。

## 2.12 答案如下:

- a) 

```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
    sum += count;
```
- b) 

```
count << setiosflags(ios::fixed | ios::showpoint | ios::left)
<< setprecision( 1 ) << setw( 15 ) << 333.546372
<< setprecision( 2 ) << setw( 15 ) << 333.546372
<< setprecision( 3 ) << setw( 15 ) << 333.546372
<< endl
```

输出结果:

333.5    333.55    333.546

- c) 

```
cout << setiosflags( ios::fixed | ios::showpoint )
<< setprecision( 2 ) << setw( 10 ) << pow( 2.5, 3 )
<< endl;
```

输出结果:

15.63

- d) 

```
x = 1;
while ( x <= 20 ) {
    cout << x;
    if ( x% 5 == 0 )
        cout << endl;
    else
        cout << '\t';
    x++;
}
```
- e) 

```
for ( x = 1; x <= 20; x++ ) {
    cout << x;
```

```

if ( x % 5 == 0 )
    cout << endl;
else
    cout << 't';

```

或者:

```

for ( x = 1; x <= 20; x++ )
    if ( x % 5 == 0 )
        cout << x << endl;
    else
        cout << x << 't';

```

### 2.13 答案如下:

- a) 错误:while 头之后的分号会导致无限循环。

改正:将分号替换成一个!,或同时删除;和!。

- b) 错误:试图用一个浮点数来控制 for 重复结构。

改正:使用一个整数,并执行正确的计算,以获得你所希望的值

```

for ( v = 1; v != 10; v++ )
    cout << ( static_cast< double> ( v ) * 10 ) << endl;

```

- c) 错误:在用于第一个 case 的语句中遗漏了 break 语句。

改正:在第一个 case 的语句末尾添加 break。注意假如程序员希望每次执行 case 1; 语句时,总是执行 case 2; 语句,便不是错误。

- d) 错误:在 while 重复继续条件中关系操作符的使用不当。

改正:用 <= 而不用 <,或把 10 改为 11。

### 练习题

练习题 2.14 ~ 2.38 与 2.1 ~ 2.12 节中的内容对应。练习题 2.39 ~ 2.63 与 2.13 ~ 2.21 节中的内容对应。

#### 2.14 找出并改正以下各代码段中的错误:

- a) `if ( age >= 65 );`  
`cout << "Age is greater than or equal to 65" << endl;`  
`else`  
`cout << "Age is less than 65" << endl;`
- b) `is ( age >= 65 )`  
`cout << "Age is greater than or equal to 65" << endl;`  
`else;`  
`cout << "Age is less than 65" << endl;`
- c) `int x = 1, total;`  
`while ( x != 10 ) {`  
`total += x;`  
`--x;`  
`}`
- d) `while ( x <= 100 )`

```

        total += x;
        ++x;
e) while ( y > 0 ){
    cout << y << endl;
    ++y;
}

```

2.15 指出下列程序的打印输出结果。

```

1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int main()
7  |
8      int y,x=1,total=0;
9
10     while (x<=10){
11         y=x*x;
12         cout<<y<<endl;
13         total+=y;
14         ++x;
15     }
16
17     cout<<"Total is"<<total<<endl;
18     return 0;
19 ;

```

针对练习题 2.16 ~ 2.19, 执行以下步骤:

- a) 阅读问题陈述。
- b) 使用伪代码和自上而下求精法制定算法。
- c) 写一个 C++ 程序。
- d) 测试、调试及执行 C++ 程序。

2.16 每位司机都关心车辆的耗油情况。有位司机记录了自己行驶的英里数, 以及每次加油多少加仑。请设计一个程序, 要求输入行驶的英里数, 以及每次加了多少加仑汽油。程序应计算并显示每次加油后, 每加仑能够行驶的英里数。处理完所有输入信息后, 程序还应综合所有输入, 计算并打印每加仑可供行驶多少英里。输出结果如下:

```

Enter the gallons used ( -1 to end): 12.8
Enter the miles driven: 287
The miles /gallon for this tank was 22.421875

Enter the gallons used ( -1 to end) : 10.3
Enter the miles driven: 200
The miles /gallon for this tank was 19.417475

Enter the gallons used ( -1 to end): 5
Enter the miles driven: 120

```

```
The miles /gallon for this tank was 24.000000
```

```
Enter the gallons used (-1 to end): -1
```

```
The overall average miles /gallon was 21.601423
```

- 2.17 开发一个C++ 程序,用于判断百货公司的客户是否超出了赊欠账户的信用额度。每一名客户,都可使用信息:

- 账号(一个整数,account number)。
- 每月开始时的欠款(beginning balance)。
- 该客户当月购买的所有商品总额(total charges)。
- 客户账户当月在此账户上存入的金额(total credits)。
- 允许的信用额度(credit limit)。

程序应要求用户输入以上每一项信息,计算新的欠款(=初始欠款+当月消费额-存入金额),并判断新的欠款额是否超过客户的信用额度(最多允许的欠款额)。对那些已经超支的客户,程序应显示客户的账户编号、信用额度、新欠款额以及“Credit Limit Exceeded”(超过信用额度)消息。

```
Enter account number (-1 to end):100
```

```
Enter beginning balance:5894.78
```

```
Enter total charges:1000.0
```

```
Enter total credits:500.00
```

```
Enter credit limit:5500.00
```

```
Account:      100
```

```
Credit limit:5500.00
```

```
Balance:      5894.78
```

```
Credit Limit Exceeded.
```

```
Enter account number(-1 to end):200
```

```
Enter beginning balance:1000.00
```

```
Enter total charges:123.45
```

```
Enter total credits:321.00
```

```
Enter credit limit:1500.00
```

```
Enter account number (-1 to end):300
```

```
Enter beginning balance:500.00
```

```
Enter total charges:274.73
```

```
Enter total credits:100.00
```

```
Enter credit limit:800.00
```

```
Enter account number (-1 to end): -1
```

- 2.18 一家大型化工厂采用佣金方式为推销员付酬。推销员每周领到基本工资 200 美元,再加上一周销售毛利的 9%。例如,一名推销员在某一周销售了毛利为 5 000 美元的化工产品,那么除领取固定的 200 美元之外,还要加上 5 000 美元的 9%,总计 650 美元。开发一个C++ 程序,用于输入推销员上一周的毛利,然后计算并显示那名推销员的收入。每次处理一名推销员的数据。

```
Enter sales in dollars ( -1 to end):5000.00
Salary is:$ 650.00
```

```
Enter sales in dollars ( -1 to end):6000.00
Salary is:$ 740.00
```

```
Enter sales in dollars ( -1 to end):7000.00
Salary is:$ 830.00
```

```
Enter sales in dollars ( -1 to end):-1
```

- 2.19 开发一个C++ 程序,判断每名员工的薪金总额。公司规定,每名员工在其工作的前 40 小时内,每小时都领取固定工资。超出 40 小时后,每工作一小时,算一个半小时。你将得到一份公司员工列表,每名员工上周工作小时数,以及每名员工每小时的工资金额。你的程序应为每名员工输入这些信息,计算并显示员工上一周的薪金总额。

```
Enter hours worked ( -1 to end):39
Enter hourly rate of the worker ( $ 00.00):10.00
Salary is #390.00
```

```
Enter hours worked ( -1 to end):40
Enter hourly rate of the worker ( $ 00.00):10.00
Salary is #40.00
```

```
Enter hours worked ( -1 to end):41
Enter hourly rate of the worker ( $ 00.00):10.00
Salary is $ 415.00
```

```
Enter hours worked ( -1 to end):-1
```

- 2.20 在计算机应用程序中,经常会用到查找最大数的操作(一组数字中的最大值)。例如,程序可输入每名推销员的销售量,从而判断一次销售比赛中的优胜者。销售量最大的人将赢得这次比赛。请写一个伪代码程序,然后将其转变成为一个C++ 程序,用它输入 10 个数字,判断并打印最大数。提示:你的程序应用到 3 个变量,如下:  
 counter:能计数到 10 的计数器(用于判断输入了多少个数字,并判断何时处理完 10 个数字)。  
 number:输入程序的当前数字。  
 largest:迄今为止找到的最大数。

- 2.21 写一个C++ 程序,利用循环和制表位换码序列\t 打印以下值表:

N	10 * N	100 * N	1000 * N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

- 2.22 采用练习题 2.20 的类似方法,找出 10 个数字中的两个最大值。注意:每个数字只能

输入一次。

2.23 修改图 2.11 中的程序,对其输入进行校验。针对每个输入,假如输入的值不是 1 或 2,那么继续循环,直到用户输入一个正确的值为止。

2.24 以下程序的打印结果是什么?

```
1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int main()
7  {
8      int count = 1;
9
10     while (count <= 10){
11         cout << (count % 2 ? " * * * * "; " + + + + + ")
12             << endl;
13         ++count;
14     }
15
16     return 0;
17 }
```

2.25 以下程序的打印结果是什么?

```
1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int main()
7  {
8      int row = 10, column;
9
10     while (row >= 1){
11         column = 1;
12
13         while (column <= 10){
14             cout << (row % 2 ? "<"; ">");
15             ++column;
16         }
17
18         --row;
19         cout << endl;
20     }
21
22     return 0;
23 }
```

2.26 (“摇摆的 else”问题)在(x 等于 9,y 等于 11)和(x 等于 11,y 等于 9)这两种情况下,判断以下每段代码的输出。注意编译器会忽略一个 C++ 程序中的所有缩排样式。另外,



C++ 编译器总是将 else 同上一个 if 关联起来,除非用一对花括号 {} 告诉它不要这么做。由于初看上去,程序员可能无法确定一个 else 到底匹配的是哪一个 if,所以把它称为“摇摆的 else”问题。我们已从下述代码中消除了所有缩排样式,目的是加大该问题的难度(提示:请应用所学的缩进规范)。

```
a) if ( x < 10 )
    if ( y > 10 )
        cout << " * * * * *" << endl;
    else
        cout << "#####" << endl;
    cout << " $ $ $ $ $" << endl;

b) if ( x < 10 ) |
    if ( y > 10 )
        cout << " * * * * *" << endl;
    |
    else {
        cout << "#####" << endl;
        cout << " $ $ $ $ $" << endl;
    }
|
```

2.27 (另一个“摇摆的 else”问题)修改以下代码,产生指定的输出。注意使用正确的缩排方法。除了插入花括号之后,你不能进行其他任何修改。编译器会忽略C++ 程序中的所有缩排。我们已从下述代码中消除了所有缩排样式,目的是加大该问题的难度(提示:有可能无需任何修改)。

```
if ( y == 8 )
if ( x == 5 )
    cout << " @ @ @ @ @ " << endl;
else
    cout << "#####" << endl;
    cout << " $ $ $ $ $" << endl;
    cout << " & & & & " << endl;
```

a) 假定  $x = 5, y = 8$ , 产生以下输出:

```
@ @ @ @ @
$ $ $ $ $
& & & & &
```

b) 假定  $x = 5, y = 8$ , 产生以下输出:

```
@ @ @ @ @
```

c) 假定  $x = 5, y = 8$ , 产生以下输出:

```
@ @ @ @ @
& & & & &
```

d) 假定  $x = 5, y = 7$ , 产生以下输出。注意, else 之后的 3 个输出语句均是一个复合语句的组部分:

```
. # # # # #
$ $ $ $ $
& & & & &
```

- 2.28 写一个程序,读取一个正方形的边长,然后利用星号和空格,打印具有那个边长的一个空心正方形。你的程序应生成边长在1~20之间的所有正方形。假定输入5,则输出效果应该像下面这样:

```
*****
*   *
*   *
*   *
*   *
*****
```

- 2.29 “回文”是一种特殊的数字或文字短语,无论顺读,还是倒读,结果都一样。例如,下面这些整数其实都是“回文”:12 321,55 555,45 554 以及 11 611。请写一个程序,读入一个5位整数,判断它是否为回文(提示:使用除法和求模操作符将数字分解为单独的数位)。
- 2.30 要求用户输入一个二进制整数,其中只应包括0和1(即一个“二进制”整数),然后把它转换成十进制打印出来。提示:按从右到左的顺序,用求模和除法操作符分离出各个数位。大家知道,在十进制数字系统中,最右侧数位的位置值是1(10的0次方);倒数第二个数位的位置值是10(10的1次方);然后是100(10的2次方),1 000(10的3次方)……等等。类似地,在二进制数字系统中,最右侧数位的位置值是1(2的0次方),倒数第二个数位的位置值是2(2的1次方),然后是4(2的2次方),8(2的3次方)……等等。因此,十进制234可以解释成: $4 * 1 + 3 * 10 + 2 * 100$ ;而二进制1 101要转换成十进制数,可解释成: $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ ,或者 $1 + 0 + 4 + 8$ ,结果为13)。
- 2.31 写一个程序,显示以下棋盘图案:

```
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
```

你的程序只能使用以下3种输出语句

```
cout << " * ";
cout << '\n';
cout << endl;
```

- 2.32 写一个程序,不断打印整数2的倍数,即2,4,6,8,16,32,64等等。你的循环不能中止(换言之,要求你创建一个无限循环)。运行这个程序会发生什么情况?
- 2.33 写一个程序,读取一个圆的半径(采取double类型),计算并打印直径、周长和面积。 $\pi$ 值为3.141 59。
- 2.34 以下语句有何错误? 提供正确的语句,以正确实现程序员的意图。

```
cout << ++( x + y );
```

- 2.35 写一个程序,读取3个非零的double值,判断并打印它们是否能代表一个三角形的3个边长。

- 2.36 写一个程序,读取 3 个非零的整数,判断并打印它们是否能代表一个直角三角形的 3 个边长。
- 2.37 一家公司想通过电话传输数据,但担心电话被人窃听。他们的所有数据都采用 4 数位整数的方式传送。现在,他们要求你写一个程序,对其数据进行加密,以便数据的传送更安全地传送。你的程序应读取一个包含 4 个数位的整数,并按以下方式加密:将每个数位替换成(那个数位加 7)求模 10。然后,让第一个和第三个数位调换位置,将第二个和第 4 个调换位置,并打印出加密后的整数。另写一个程序,输入一个加密的 4 位整数,解密还原成原先的数字。
- 2.38 对一个非负整数  $n$  来说,它的阶乘可写成  $n!$  (读作“ $n$  阶”)。假如  $n$  大于或等于 1,则计算公式如下:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

假如  $n$  等于 0,则计算公式如下:

$$n! = 1$$

例如,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , 结果为 120。

- a) 写一个程序,要求用户输入一个非负整数,计算并打印阶乘结果。
- b) 写一个程序,用公式

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

估算数学常量  $e$  的值。

- c) 写一个程序,计算  $e^x$ , 公式如下

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

- 2.39 指出以下各段代码中的错误:

- a) `for ( x = 100, x >= 1, x++ )`

`cout << x << endl;`

- b) 以下代码应打印整数 `value` 是奇数还是偶数:

`switch ( value % 2 ) {`

`case 0:`

`cout << "Even integer" << endl;`

`case 1:`

`cout << "Odd integer" << endl;`

`}`

- c) 代码

`for ( x = 19; x >= 1; x += 2 )`

`cout << x << endl;`

将输入 19 ~ 1 之间的奇数。

- d) 代码

`counter = 2;`

`do {`

`cout << counter << endl;`

`counter += 2;`

```
| While ( counter < 100 );
```

将输出 2 ~ 100 之间的偶数。

- 2.40 写一个程序,对一系列整数求和。假定读入的第一个整数指定了后续还要输入多少个值。在你的程序中,每个输入语句只应读入一个值。一个典型的输入序列

```
5 100 200 300 400 500
```

中,5 代表随后对 5 个值求和。

- 2.41 写一个程序,计算并打印几个整数的平均值。假定读入的最后一个值是标记 9 999。一个典型的输入序列

```
10 8 11 7 9 9999
```

表明为 9 999 之前的所有值求平均值。

- 2.42 指出以下程序的用途。

```
1 #include <iostream>
2
3 using std::cout;
4 using std::cin;
5 using std::endl;
6
7 int main()
8 |
9     int x, y;
10
11     cout << "Enter two integers in the range 1-20:";
12     cin >> x >> y;
13
14     for (int i = 1; i <= y; i++) |
15
16         for (int j = 1; j <= x; j++)
17             cout << '@'
18
19             cout << endl;
20     |
21
22     return 0;
23 |
```

- 2.43 写一个程序,找出几个整数的最小值。假定读入的第一个值指定了后续将输入多少个值,第一个数字不在要计算的整数之列。
- 2.44 写一个程序,计算和打印 1 ~ 15 之间所有奇数的乘积。
- 2.45 阶乘函数常用于概率问题。一个正整数  $n$  的阶乘(写成  $n!$ ,读作“ $n$  阶”)等于从 1 到  $n$  的正整数乘积。写一个程序,计算从 1 ~ 5 之间各个整数的阶乘。以表格形式打印结果。计算 20 的阶乘有在哪些难处?
- 2.46 修改 2.15 节的复利程序,分别以 5%,6%,7%,8%,9% 和 10% 的利率计算结果。用一个 for 循环更改利率。
- 2.47 写一个程序,打印以下图案。每个图案都接在上一个图案的下方。用 for 循环生成图案。所有星号(\*)都用单条语句打印,形如 `cout << ' *';`(这会导致星号紧挨)。提



计算  $\pi$  的值。打印一个表格,分别显示计算 1 项、2 项、3 项……时, $\pi$  的近似值。这个数列要计算到多少项,才能得到  $\pi$  的近似值为 3.14? 3.141? 3.141 5? 3.141 59?

- 2.55 (勾股定理,国外称“毕格拉斯定理”) 对一个直角三角形来说,我们可让它的所有边长都为整数。在此,我们把这 3 个整数称作“毕格拉斯三元组”(Pythagorean triple)。这三边肯定满足这样一个关系:两个直边(side1 和 side2,分别对应“勾”和“股”)的平方和等于斜边(hypotenuse,对应“弦”)的平方。请写一个程序,找出 side1、side2 和 hypotenuse 值不大于 500 的所有毕格拉斯三元组。请用一个三级嵌套的 for 循环来尝试所有可能性。注意这是一种很典型的“野蛮算法”。日后学习更高级的编程课程时,还会了解有关此类问题更有趣的解法。
- 2.56 一家公司对员工进行分类,分类计算工资。分为经理(每周固定工资);计时工(前 40 小时,按固定工资计;超出 40 小时,每小时算 1.5 小时);佣金工(每周除固定 250 元工资外,再加销售毛利的 5.7%);或者计件工(根据产品件数,发固定工资,每名计件工都只生产一种类型的产品)。写一个程序,计算每个员工的周薪。假设事先并不知道员工数量。每种类型的员工都有自己的工资代码:经理的工资代码是 1,小时工的代码是 2,佣金工的代码是 3,而计件工的代码是 4。请用一个 switch 结构根据员工的工资代码计算他们的工资。在 switch 内部,提醒用户(会计)输入计算每一名员工工资所需的一系列基本事实。
- 2.57 (摩根定律)本章讨论了逻辑操作符 &&,|| 和 !。利用摩根定律,有时可以更方便地表示一个逻辑表达式。这些定律指出,表达式!(条件1 && 条件2)在逻辑上相当于表达式(!条件1 || !条件2)。此外,表达式!(条件1 || 条件2)在逻辑上相当于表达式(!条件1 && !条件2)。请根据摩根定律,写出同下述表达式对应的表达式。然后写一个程序,同时显示在每一种情况下,对应的原始表达式和新表达式:
- a) ! ( x < 5 ) && ! ( y >= 7 )
- b) ! ( a == b ) || ! ( g != 5 )
- c) ! ( ( x <= 8 ) && ( y > 4 ) )
- d) ! ( ( i > 4 ) || ( j <= 6 ) )
- 2.58 写一个程序,打印以下菱形。你可让输出语句要么打印一个星号(\*),要么打印一个空格。尽量多用重复(使用嵌套 for 结构),避免使用输出语句。

```

      *
     ***
    *****
   *********
  ***********
 *****
  *****
   *****
    *****
     ***
      *

```

- 2.59 修改你为练习题 2.58 写的程序,令其读取 1~19 之间的一个奇数,指定菱形中的行数。然后,你的程序应显示出大小合适的一个菱形。
- 2.60 有人认为 break 和 continue 语句是非结构化的。事实上,break 和 continue 语句总能用结构化的语句替代,尽管这样做显得过于笨拙。解释如何从程序的一个循环中删除所

有 break 语句,并替换成结构化的语句。提示:break 语句用于在循环主体内离开一个循环。为了离开循环,另一个办法是让循环继续测试失败。考虑在循环继续测试中使用另一个测试,指出“由于符合一个‘break’条件,所以提前退出”。利用这里介绍的技术删除图 2.26 所示程序中的 break 语句。

2.61 指出以下程序段的用途?

```
1  for ( i = 1; i <= 5; i++ ) {
2    for ( j = 1; j <= 3; j++ ) {
3      for ( k = 1; k <= 4; k++ )
4        cout << '*';
5      cout << endl;
6    }
7    cout << endl;
8  }
```

2.62 解释如何从程序的一个循环中删除任何 continue 语句,并将其替换为某种对应的结构化语句。利用这里介绍的技术,从图 2.27 所示程序中删除 continue 语句。

2.63 (“圣诞十二天”之歌)写一个有趣的程序,利用重复和 switch 结构打印“圣诞十二天”(The Twelve Days of Christmas)这首歌。应该用一个 switch 结构打印天数(即 first, second 等等)。另一个 switch 结构用于打印剩余的歌词。歌词如下:

On the first day of Christmas my true love sent to me: A Partridge in a Pear Tree	Three French Hens Two Turtle Doves and a Partridge in a Pear Tree
On the second day of Christmas my true love sent to me: Two Turtle Doves and a Partridge in a Pear Tree	On the sixth day of Christmas my true love sent to me: Six Geese a Laying Five Golden Rings Four Calling Birds Three French Hens Two Turtle Doves and a Partridge in a Pear Tree
On the third day of Christmas my true love sent to me: Three French Hens Two Turtle Doves and a Partridge in a Pear Tree	On the seventh day of Christmas my true love sent to me: Seven Swans a Swimming Six Geese a Laying Five Golden Rings Four Calling Birds Three French Hens Two Turtle Doves and a Partridge in a Pear Tree
On the fourth day of Christmas my true love sent to me: Four Calling Birds Three French Hens Two Turtle Doves and a Partridge in a Pear Tree	On the eighth day of Christmas my true love sent to me: Eight Maids a Milking
On the fifth day of Christmas my true love sent to me: Five Golden Rings Four Calling Birds	

Seven Swans a Swimming  
Six Geese a Laying  
Five Golden Rings  
Four Calling Birds  
Three French Hens  
Two Turtle Doves  
and a Partridge in a Pear Tree

On the ninth day of Christmas  
my true love sent to me;  
Nine Ladies Dancing  
Eight Maids a Milking  
Seven Swans a Swimming  
Six Geese a Laying  
Five Golden Rings  
Four Calling Birds  
Three French Hens  
Two Turtle Doves  
and a Partridge in a Pear Tree

On the tenth day of Christmas  
my true love sent to me;  
Ten Lords a Leaping  
Nine Ladies Dancing  
Eight Maids a Milking  
Seven Swans a Swimming  
Six Geese a Laying  
Five Golden Rings  
Four Calling Birds  
Three French Hens  
Two Turtle Doves

and a Partridge in a Pear Tree

On the eleventh day of Christmas  
my true love sent to me;  
Eleven Pipers Piping  
Ten Lords a Leaping  
Nine Ladies Dancing  
Eight Maids a Milking  
Seven Swans a Swimming  
Six Geese a Laying  
Five Golden Rings  
Four Calling Birds  
Three French Hens  
Two Turtle Doves  
and a Partridge in a Pear Tree

On the twelfth day of Christmas  
my true love sent to me;  
12 Drummers Drumming  
Eleven Pipers Piping  
Ten Lords a Leaping  
Nine Ladies Dancing  
Eight Maids a Milking  
Seven Swans a Swimming  
Six Geese a Laying  
Five Golden Rings  
Four Calling Birds  
Three French Hens  
Two Turtle Doves  
and a Partridge in a Pear Tree

练习题 2.64 与 2.22 节“对象思考”中的内容对应。

- 2.64 用 200 字描述汽车的定义及功能。分别列出你用到的名词和动词。在书中,我们曾说过每个名词都有可能代表一个对象,需要构建它以实现一个系统(目前是汽车系统)。请你选出其中的 5 个对象,分别指出它们的几种属性和行为。简要说明这些对象同你描述的其他对象如何交互。这其实正是一个典型的面向对象设计中涉及的几个关键步骤。
- 2.65 (Peter Minuit 问题)据说, Peter Minuit 于 1626 年花 24.00 美元购买了曼哈顿城。这笔投资合算吗? 为解答这个问题,请修改图 2.21 的复利程序,将本金设为 24.00 美元,计算到今年为止(截止 2002 年,总共 376 年),利滚利算下来的存款总金额。分别设定利率为 5%,6%,7%,8%,9% 和 10%,体验复利的奇迹!



# 第3章 函 数

## 学习目标

- 理解如何以名为“函数”的小代码块为基础,用模块化方式构建程序
- 会创建新函数
- 理解函数间的信息传递机制
- 会利用随机数生成机制,实现模拟技术
- 理解标识符如何被限定在特定的程序区域中
- 理解如何编写和使用调用其自身的函数

## 3.1 简介

解决现实问题的大多数计算机程序都比前两章介绍的程序复杂。经验表明,开发和维护大型程序时,最好的办法便是以较小的部分或组件为基础进行构建。同原始程序相比,类似的每个小组件都要好管理一些。我们将这种技术称为“分而治之”。本章将对C++ 语言的许多关键特性进行讲解。利用这些特性,可以更方便地设计、实现、运行和维护大型程序。

## 3.2 C++ 中的程序组件

C++ 中的模块叫作“函数”和“类”。程序员通常将自己写的新函数同C++ 标准库中提供的“预打包”函数合并到一起,并将自己写的新类同各个类库提供的“预打包”类合并到一起,构建出完整的C++ 程序。在本章,我们将把重点放在函数上;我们将从第6 章起,讨论类的细节。

C++ 标准库提供了丰富的内建函数(这些函数用于执行常见的数学计算、字符串处理、字符处理、输入/输出、错误检查以及其他许多有用的操作。利用它们,程序员的工作可以变得更轻松,因为这些函数提供了程序员所需的许多功能。C++ 标准库函数是C++ 编程环境中牢不可分的组成部分。

**良好编程习惯 3.1** 尽快掌握C++ 标准库中的函数和类集合。

**软件工程知识 3.1** 避免万事从头开始。在可能的情况下,尽量使用C++ 标准库中的函数,而不是从头编写新函数。这样可减少程序开发时间。

**可移植性提示 3.1** 利用C++ 标准库中的函数有助于提高程序的可移植性。

**性能提示 3.1** 对于现有的库程序,不要为了使其更有效而试图重写它。这些程序的性能通常已被发挥得淋漓尽致,没有提升的余地。

程序员可编写一些函数,以便定义特定的任务,并可多次用于程序中。我们把它称作“程序员自定义函数”。要注意的是,用来定义函数的实际语句只需编写一次。而且在其他函数面前,这些语句是“隐身”的(调用时,根本不必关心具体语句)。

函数被“函数调用”来“唤醒”(亦即让它执行目标任务)。在函数调用中,需要指定函数的名称,并提供被调用函数执行任务时所需的相应信息(比如参数等等)。事实上,这和公司里的分级管理系统非常相似。老板(也就是“调用函数”、“主叫者”或“调用者”)要求一名员工(也就是“被调用函数”)完成一项任务,并在完成后返回(汇报)结果。在这个过程中,老板函数并不知道员工函数具体是如何完成任务的。员工完全可能再调用其他员工函数,老板根本不会注意到这一点。大家不久便会见识到如何利用这种实现细节的“隐藏”,来优化软件工程。在图 3.1 展示了一个程序(老板 - boss)如何利用分级方式同几个员工函数通信。注意 worker1 在此又扮演着 worker4 和 worker5 的老板函数的角色。当然,函数之间的实际关系可能会和这张图中显示的分级结构有所区别。

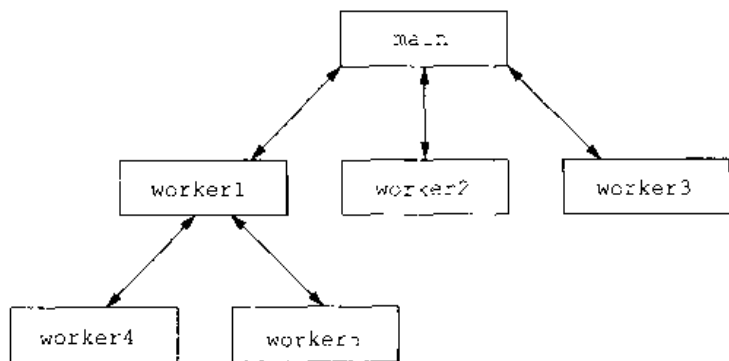


图 3.1 分级式老板函数/员工函数的关系

### 3.3 数学库函数

利用数学库函数,程序员可以执行一些普通的数学计算。在此,我们打算利用各类不同数学库函数来介绍函数的概念。本书稍后还将讨论 C++ 标准库中的许多其他函数。

调用函数时,通常要写下它的名称,再在名称后添加一对括号(圆括号),括号内写下该函数所需的参数或自变量(多个参数用逗号分隔)。例如,假如一个程序员打算计算并打印 900.0 的平方根,就得采用语句

```
cout << sqrt( 900.0 );
```

执行该语句时,会调用数学库函数 sqrt 来计算圆括号(900.0)内包含的数字的平方根。其中,900.0 便是 sqrt 的参数。前面的语句将打印 30。sqrt 函数采用了 double 类型的参数,并返回 double 类型的结果。数学库函数中的所有函数都会返回 double 数据类型的结果。为了使用数学库函数,应将 <cmath> 头文件包含在程序当中。

**常见编程错误 3.1** 使用数学库函数时,忘记将数学头文件包含在内是语法错误。程序中所用的每一个标准库函数都必须包含标准的头文件。

函数参数可以是常量、变量或表达式。例如,假如  $c1 = 13.0$ ,  $d = 3.0$ , 而  $f = 4.0$ , 那么

语句

```
cout << sqrt( c1 + d * f );
```

将计算并打印  $13.0 + 3.0 * 4.0 = 25.0$  的平方根,结果等于 5(这是因为在 C++ 中,对于无小数部分的浮点数,一般不打印其小数点以及后面的 0)。

图 3.2 总结了一些常用的数学库函数。该图中,变量  $x$  和  $y$  均为 `double` 类型。

函数	说明	示例
<code>ceil( x )</code>	求 $x$ 的整数部分,使其不小于 $x$ 的最小整数	<code>ceil( 9.2 )</code> 等于 10.0
<code>cos( x )</code>	求 $x$ 的余弦( $x$ 用弧度表示)	<code>cos( 0 )</code> 等于 1
<code>exp( x )</code>	求 $e$ 的 $x$ 次方	<code>exp( 1 )</code> 等于 2.718 28, <code>exp( 2 )</code> 等于 7.389 06
<code>abs( x )</code>	求 $x$ 的绝对值	<code>abs( 5.1 )</code> 等于 5.1, <code>abs( 0 )</code> 等于 0, <code>abs( -8.76 )</code> 等于 8.76
<code>floor( x )</code>	求 $x$ 的整数部分,使其不大于 $x$ 的最大整数	<code>floor( 9.2 )</code> 等于 9.0
<code>fmod( x, y )</code>	$x/y$ 的浮点数余数	<code>fmod( 13.657, 2.333 )</code> 等于 1.992
<code>ln( x )</code>	求 $x$ 的自然对数(以 $e$ 为底数)	<code>ln( 2.718 282 )</code> 等于 1, <code>ln( 7.389 056 )</code> 等于 2
<code>lg10( x )</code>	求 $x$ 的对数(以 10 为底数)	<code>lg10( 10.0 )</code> 等于 1.0, <code>lg10( 100.0 )</code> 等于 2.0
<code>pow( x, y )</code>	求 $x$ 的 $y$ 次方	<code>pow( 2, 7 )</code> 是 128, <code>pow( 9, .5 )</code> 等于 3
<code>sin( x )</code>	求 $x$ 的正弦( $x$ 用弧度表示)	<code>sin( 0 )</code> 等于 0
<code>sqrt( x )</code>	求 $x$ 的平方根	<code>sqrt( 900 )</code> 等于 30, <code>sqrt( 9 )</code> 等于 3
<code>tan( x )</code>	求 $x$ 的正切	<code>tan( 0.0 )</code> 等于 0

图 3.2 常用数学库函数

## 3.4 函数

有了函数,程序员可以使程序模块化。函数定义中声明的所有变量都是“局部变量”(local variables)——只能用于定义了它们的函数中。许多函数都有一个“参数”列表,提供了函数之间沟通信息的方式。函数的参数也是局部变量。

**软件工程知识 3.2** 在包含许多函数的程序中,,main 应被视作一组函数调用来执行,这组函数用于执行程序的大部分工作。

有几方面的原因促使我们要对一个程序进行“功能化”(亦即把程序分成相互作用的函数)。首先,利用这种“分而治之”的技术,程序的开发过程可变得更易管理。其次,这样做有利于保证软件的“重用性”——即利用现成的函数来构建新程序。在面向对象的编程实践中,软件的重用性是至关重要的环节。采用良好的函数命名和定义,我们可把用于完成特定任务的函数组合到一起,最终构建出一个崭新的程序,而不是利用自定义代码来构建。最后一个原因是这样做可避免在程序里出现重复的代码。把代码打包成函数以后,这些代码就可以在程序中多个位置执行,届时只需调用函数即可,无需原样照抄。

**软件工程知识 3.3** 每个函数都应限于只完成一个单独的、具有良好定义的任务,而且函数名称应该更能有效地代表它所完成的任务。这样可以提升软件的重用性。

**软件工程知识 3.4** 如果实在无法想出能准确表达函数作用的名称,则表明你定义的函数执行的任务可能太分散。碰到类似情况,最好把这类函数分成几个较小的函数,令其各自负责某项特定任务。

## 3.5 函数定义

前面介绍的程序都使用了 main 函数,该函数调用了标准库函数来完成自己的任务。现在,我们要考虑程序员应如何编写自己的自定义函数。

以这样的程序为例,它采用用户自定义函数 square 来计算 1~10 之间的整数的平方根(如图 3.3 所示)。

```

1 //Fig.3.3: fig03_03.cpp
2 //Creating and using a programmer-defined function
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int square( int ); //function prototype
9
10 int main()
11 {
12     for ( int x = 1; x <= 10; x++ )
13         cout << square( x ) << " ";
14
15     cout << endl;
16     return 0;
17 }
18
19 //Function definition
20 int square( int y )
21 {
22     return y * y;
23 }
```

输出结果:

```
1 4 9 16 25 36 49 64 81 100
```

图 3.3 创建和使用程序员自定义的函数

**良好编程习惯 3.2** 在函数定义之间放置一个空行,以分隔函数并增强程序的可读性。

在 main 函数中,函数 square 的调用

```
square( x )
```

函数 square 在参数 y 中收到了 x 值的副本。然后 square 对 y \* y 进行计算。计算结果被传回 main 函数中调用 square 函数的位置,并显示结果。注意,x 的值不会被函数调用改变。该过程将用 for 重复结构重复执行 10 次。

square 函数的定义表明它需要一个整数参数 y。函数名前面的关键字 int 表明 square 函

数将返回一个整数结果。square 中的 return 语句将计算结果返回调用函数。

第 8 行

```
int square( int ); //function prototype
```

是一个函数原型。圆括号中的 int 数据类型是告诉编译器,函数 square 需要调用者提供一个整数值。函数名 square 名称旁边的 int 数据类型则告诉编译器,函数 square 将向调用者返回一个整数结果。编译器利用函数原型来检查 square 调用中是否包含了准确的返回类型、准确的参数数目、参数类型和参数出现的顺序是否正确。如果函数定义出现在程序中首次使用该函数之前,就不需要其函数原型。类似情况下,函数定义也可以充当函数原型。如果图 3.3 中的第 20~23 行之间的函数出现在 main 之前,第 8 行中的函数原型就可以省略。关于函数原型的详细讨论,请参见 3.6 节。

函数定义的格式如下

```
return-value-type function-name( parameter-list )
{
    declarations and statements
}
```

function-name(函数名)是任意一个有效的标识符。return - value - type(返回值的类型)是从函数返回调用者的计算结果的数据类型。返回值类型 void 表明函数没有返回值。

**常见编程错误 3.2** 函数定义中省略返回值类型是语法错误。

**常见编程错误 3.3** 忘记在需要返回值的函数中返回返回值是语法错误。

**常见编程错误 3.4** 在返回值类型已被声明为 void 的函数中返回返回值是语法错误。

parameter - list(参数列表)是一个用逗号隔开的列表,其中包含函数在被调用时收到的参数。如果函数不接受任何值,参数列表将为 void 或简单地留空。对于函数参数列表中的每个参数而言,其类型都必须显式指定。

**常见编程错误 3.5** 将同一类型的函数参数声明为 float x,y,而不是 float x,float y。参数声明 float x,y 实际上会报告编译出错。因为参数列表中每个参数都需要指明类型。

**常见编程错误 3.6** 在函数定义的参数列表右括号之后添加分号是语法错误。

**常见编程错误 3.7** 将函数参数再次定义为函数中的局部变量是语法错误。

**良好编程习惯 3.3** 尽管不会错;但最好不要使传递给函数的形参与函数定义中的实参同名。这样可以避免歧义性。

**常见编程错误 3.8** 函数调用中的圆括号()实际上是C++中的操作符。它可以使函数被调用。在不采用参数函数调用中,忘记使用圆括号()并不是语法错误。但在你打算执行函数调用时,函数可能无法被调用。

花括号内的 declarations(声明)和 statements(语句)形成了函数主体(function body)。函数主体也被成为“块”。块是一个简单的、其中包含声明的复合语句。变量可以在任何一个块中进行声明,而且块还可以实行嵌套。在任何情况下,函数都不能在另一个函数中定义。

**常见编程错误 3.9** 在一个函数中定义另一个函数是语法错误。

**良好编程习惯 3.4** 选择含义明确的函数名称和参数名可以使程序更具可读性,而且还有助于避免使用大量的注释。

**软件工程知识 3.5** 函数应该适应编辑器窗口的大小。不管函数有多长,它都应该能很好地完成任务。小函数有助于提升软件的重用性。

**软件工程知识 3.6** 程序应该写为若干小函数的集合。这样可以使程序更易于编写、调试、维护和修改。

**软件工程知识 3.7** 需要大量参数的函数可能会执行大量任务。此时,可考虑把函数分解成更小的函数,令各个小函数分担个别任务。必要的情况下,函数的头部应该包含在单独的一行内。

**常见编程错误 3.10** 如果形参和实参的函数原型、函数头部和函数调用在数目、类型、出现顺序以及返回值类型不一致,就表明这是语法错误。

将控制返回函数调用位置的方式有 3 种。如果函数不返回结果,控制只能在到达函数结束时的右花括号处或执行语句

```
return;
```

时返回。如果函数不返回结果,语句

```
return expression
```

就会把表达式的值返回调用者。

我们的第二个示例用了一个程序员自定义函数 `maximum` 来判断并返回 3 个整数的最大值(如图 3.4 所示)。

```
1 //Fig.3.4: fig03_04.cpp
2 //Finding the maximum of three integers
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int maximum( int, int, int ); //function prototype
10
11 int main()
12 |
13 |   int a, b, c;
14 |
15 |   cout << "Enter three integers: ";
16 |   cin >> a >> b >> c;
17 |
18 |   //a, b and c below are arguments to
19 |   //the maximum function call
20 |   cout << "Maximum is: " << maximum( a, b, c ) << endl;
21 |
22 |   return 0;
```

```
23 |  
24  
25 //Function maximum definition  
26 //x, y and z below are parameters to  
27 //the maximum function definition  
28 int maximum( int x, int y, int z )  
29 |  
30     int max = x;  
31  
32     if ( y > max )  
33         max = y;  
34  
35     if ( z > max )  
36         max = z;  
37  
38     return max;  
39 |
```

输出结果:

```
Enter three integers:22 85 17  
Maximum is:85  
  
Enter three integers:92 35 14  
Maximum is:92  
  
Enter three integers:45 19 98  
Maximum is:98
```

图 3.4 程序员自定义的 maximum 函数

输入这 3 个整数。随后,这些整数被传递给 maximum,该函数会判断出最大值。这个值通过 maximum 中的 return 语句被返回 main。然后,返回值就会被打印出来。

## 3.6 函数原型

C++ 最重要的特性之一是“函数原型”。函数原型把函数名称、函数返回的数据类型、函数希望接收的参数数目和、参数类型和参数出现顺序告诉编译器。编译器采用函数原型来验证函数调用。C 的早期版本不能执行这类验证,所以在函数调用出错时,编译器可能无法侦测到错误。此类调用会导致致命的或非致命性的运行时错误,这些错误会导致不易确定的、难以侦测的逻辑错误。函数原型则可以纠正这一不足。

**软件工程知识 3.8** C++ 语言中,函数原型是必须的。利用#include 预处理程序指令可从相应库的头文件中获得标准库函数的函数原型。此外,使用#include 还可获得其中包含你和/或小组成员所用函数原型的头文件。

**软件工程知识 3.9** 如果函数定义出现在程序第一次使用该函数之前,就不需要函数原型。此时,函数的定义就充当了函数原型。

对图 3.4 中的 maximum 而言,其函数原型

```
int maximum( int, int, int );
```

表示 `maximum` 采用了 3 个 `int` 类型的参数,返回 `int` 类型的结果。注意,该函数原型与 `maximum` 函数定义的头文件是一样的,区别仅在于未包含参数名(`x`,`y` 和 `z`)。

**良好编程习惯 3.5** 许多程序员都用函数原型中的参数名来描述函数。编译器会忽略这些名称。

**常见编程错误 3.11** 忘记在函数原型末尾添加分号是语法错误。

函数原型中,包含函数名称及其参数类型的部分被成为“函数签名”或简单地成为“签名”。函数的签名不包含函数的返回类型。

**常见编程错误 3.12** 不符合函数原型的调用是语法错误。

**常见编程错误 3.13** 函数原型和函数定义不一致,也是语法错误。

以前面常见编程错误为例,在图 3.4 中,如果函数原型被写为

```
void maximum( int, int, int );
```

编译器就会报告出错,因为函数原型中的 `void` 返回类型与函数头文件中的 `int` 返回类型不一致。

函数原型的另一个重要特性是“强制参数类型转换”(coercion of arguments),也就是说,强制参数采用相应的类型。举个例子来说,`sqrt` 数学库函数可以用整数参数来调用,即使 `<cmath>` 中的函数原型已指定了 `double` 参数,函数仍让可以正常运行。语句

```
cout << sqrt( 4 );
```

准确无误地对 `sqrt( 4 )` 求值,并打印出结果值 2。在参数值被传递到 `sqrt` 之前,函数原型将令编译器将整数参数值 4 转换为 `double` 值 4.0。通常情况下,与函数原型中参数类型不完全一致的参数值都会在函数被调用之前,转换为正确的类型。如果不遵循 C++ 的“提升规则”,这类转换会导致不正确的结果。提升规则指定了如何在不丢失数据的前提下,实现类型的转换。在我们前面的 `sqrt` 示例中,`int` 被自动转换为 `double`,数值并没有发生变化。然而在 `double` 转换为 `int` 时,通常会截掉 `double` 值的小数部分。把大整数值类型转换为小整数类型(比如 `long` 转换为 `short`)也会导致数值的改变。

提升规则应用于其中包含 2 个以上数据类型的表达式;此类表达式被称为“混合类型表达式”。混合类型表达式中,各数值的类型被提升为表达式中的“最高”类型(实际上是创建了每个数值的临时值,并被用于表达式中,原始数值保持不变)。提升规则的另一个常见用途是用于函数参数的类型与函数定义中制定的参数类型不一致时。图 3.5 按照“最高类型”到“最低类型”的顺序,列出了内建数据类型。

把数值转换为低级数据类型会导致数值不正确。因此,只能为数值显式指定较低类型的变量或利用强制类型转换操作符,将数值转换为低级数据类型。函数参数值被转换为函数原型中的参数类型,如同它们被直接分配给这些类型的变量一样。如果采用了整数参数的 `square` 函数(如图 3.3 所示)被浮点参数所调用,那么该参数会被转换为 `int`(低级类型),而 `square` 通常会返回错误值。例如 `square( 4.5 )` 返回的数值会是 16,而不是 20.25。

**常见编程错误 3.14** 提升规则中,把高级数据类型转换为低级数据类型会更改数据值。

**常见编程错误 3.15** 函数未在初次调用之前定义时,遗漏函数原型是语法错误。



**数据类型**

long double	
double	
float	
unsigned long int	等同于 unsigned long
long int	等同于 long
unsigned int	等同于 unsigned
int	
unsigned short int	等同于 unsigned short
short int	
unsigned char	
char	
bool	(false 为 0, true 为 1)

图 3.5 内建数据类型的提升规则

**软件工程知识 3.10** 在文件中,放置于任何函数定义以外的函数原型适用于出现在函数原型之后的所有对该函数的调用。函数内部的函数原型只适用于在该函数内部执行的调用。

### 3.7 头文件

每个标准库都有相应的“头文件”(header file),头文件中包含对应库中所有函数的函数原型和这些函数所需的各种数据类型和常量的定义。图 3.6 列出了部分常见的 C++ 标准库头文件,这些文件可以包含在 C++ 程序内。图 3.6 中多次出现的“宏”将在第 17 章详细讲解。以 .h 结尾的头文件是旧式的头文件,它们已经被 C++ 标准库头文件代替。

标准库头文件	说明
<cassert>	其中包含的宏和信息用于添加有助于程序调试的诊断。
<cctype>	其中包含用于测试字符特定属性的函数原型和用于将大写字母转换为小写字母或把小写字母转换为大写字母的函数原型 <ctype.h>。
<cfloat>	其中包含系统的浮点长度限制。该头文件已经取代了 <float.h>。
<climits>	其中包含系统的整数长度限制。该头文件已经取代了 <limits.h>。
<cmath>	其中包含数学库函数的函数原型。该头文件已经取代了 <math.h>。
<cstdio>	其中包含标准输入/输出库函数的函数原型以及这些函数所用的信息。该头文件已经取代了 <stdio.h>。
<cstdlib>	其中包含把数字转换为文本、把文本转换为数字、内存分配、随机数和其他各种工具函数所用的函数原型。该头文件已经取代了 <stdlib.h>。
<cstring>	其中包含 C 类型字符串处理函数的函数原型。该头文件已经取代了 <string.h>。
<ctime>	其中包含维护时间和日期的函数原型和类型。该头文件已经取代了 <time.h>。
<iostream>	其中包含标准输入和标准输出函数的函数原型。该头文件已经取代了 <iostream.h>。
<iomanip>	其中包含可以格式化数据流的流操纵算子的函数原型。该头文件已取代了 <iomanip.h>。
<fstream>	其中包含执行磁盘文件输入和输出到磁盘文件所用的函数原型(详情参见第 14 章)。该头文件已经取代了 <fstream.h>。

标准库头文件	说明
<utility>	其中包含大部分标准库头文件所用的类和函数。
<vector>, <list>, <deque>, <queue>, < stack>, <map>, <set>, <bitset>	该头文件中包含实现标准库容器的类。容器用于在程序执行期间保存数据。我们将在标 题为“标准模板库”的一章中讨论这些头文件。
<functional>	其中包含标准库算法所用的类和函数。
<memory>	其中包含标准库所用的类和函数,以便将内存分配给标准库容器。
<iterator>	其中包含操纵标准库容器中的数据时所用的类。
<algorithm>	其中包含操纵标准库容器内的数据时所用的函数。
<exception>, <stdexcept>	这类头文件中包含用于异常处理的类(参见第13章)。
<string>	其中包含来自标准库的 string 类的定义(参见第19章)。
<sstream>	其中包含用于执行内存字符串输入和输出到内存字符串的函数的函数原型。
<locale>	其中包含数据流处理所用的类和函数,用来处理不同语言形式的数据(比如货币格式、排 序字符串、字符表示等)。
<limits>	其中包含各计算机平台定义特定数字数据类型所用的类。
<typeinfo>	其中包含用于标识运行时类型的类(在执行时确定数据类型)。

图 3.6 标准库头文件

程序员可以创建自定义的头文件。程序员自定义的头文件应该以 .h 结尾。程序员自定义的头文件可以用 #include 预处理程序指令来包含。例如,头文件 square.h 可以通过预处理程序指令

```
#include "square.h"
```

包含在程序头部。17.2 节将介绍如何包含头文件相关信息。

## 3.8 生成随机数

现在,我们要简要探讨大家都喜欢的一种“娱乐性”应用——模拟和玩游戏。在本节和下一节,我们打算开发一个结构良好的游戏程序,该程序中包含了多个函数。程序中会使用前文所述的许多控制结构。

赌博似乎是人的天性。在美国,人们喜欢前往各种赌博场所,在大大小小的、装修较为豪华的地方,以各种各样的方式,挥霍着自己的金钱。赌博是一种典型的“投机”行为,幸运者可在短时间内把一口袋钱变成一座金山。在计算机程序中,我们则用 rand 函数来模拟进行这种“投机”。

语句

```
i = rand();
```

在这里,rand 函数生成 0 到 RAND\_MAX(是 <cstdlib> 头文件中定义的一个符号常量)的一个无符号整数。RAND\_MAX 的值至少应为 32 767——一个两字节整数(即 16 位)所能表达的最大值。如果 rand 确实以随机方式生成整数值,那么在每次调用 rand 函数时,0 到 RAND\_MAX 之间的每个数出现的机会(或概率)是一样的。

调用 rand 直接生成的值往往与特定应用程序中所需要的不尽相同。例如,模拟抛硬币

的程序只要求用数字 0 表示“字”面,用数字 1 表示“徽”面;模拟掷骰子的程序只需要生成 1~6 之间的整数;而在一个假想的视频游戏中,地平线上随机出现下一类飞船(共有 4 种可能性)的程序只需要 1~4 之间的整数。

为演示 rand 函数,我们要开发一个程序,模拟掷骰子 20 次,并打印每次所得的值。rand 函数的函数原型可以在 <cstdlib> 头文件中找到。为了生成 0 到 5 之间的整数,我们采用了求模操作符(%)和 rand 函数

```
rand() % 6
```

这就叫作“比例缩放”。数字 6 被称为“比例因子”,随后,我们在前面生成的数值范围基础上加 1,获得希望的结果。图 3.7 确认了生成的结果的确在 1~6 之间。

```
1 //Fig. 3.7: fig03_07.cpp
2 //Shifted, scaled integers produced by 1 + rand() % 6
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>
13
14 int main()
15 {
16     for ( int i = 1; i <= 20; i++ ) {
17         cout << setw( 10 ) << ( 1 + rand() % 6 );
18
19         if ( i % 5 == 0 )
20             cout << endl;
21     }
22
23     return 0;
24 }
```

输出结果:

```
5  5   3   5   5
2  4   2   5   5
5  3   2   2   1
5  1   4   6   4
```

图 3.7 移动并用公式  $1 + \text{rand}() \% 6$  按比例缩放生成的整数

为了展示这些值出现的机会均等,我们利用图 3.8 中的程序模拟掷骰子 6 000 次。1~6 之间的每个整数出现的机会都应该约为 1 000 次。

```
1 //Fig. 3.8: fig03_08.cpp
2 //Roll a six-sided die 6000 times
3 #include <iostream>
4
```

```
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>
13
14 int main()
15 {
16     int frequency1 = 0, frequency2 = 0,
17         frequency3 = 0, frequency4 = 0,
18         frequency5 = 0, frequency6 = 0,
19         face;
20
21     for ( int roll = 1; roll <= 6000; roll++ ) {
22         face = 1 + rand() % 6;
23
24         switch ( face ) {
25             case 1:
26                 ++frequency1;
27                 break;
28             case 2:
29                 ++frequency2;
30                 break;
31             case 3:
32                 ++frequency3;
33                 break;
34             case 4:
35                 ++frequency4;
36                 break;
37             case 5:
38                 ++frequency5;
39                 break;
40             case 6:
41                 ++frequency6;
42                 break;
43             default:
44                 cout << "should never get here! ";
45                 |
46                 {
47
48             cout << "Face" << setw( 13 ) << "Frequency"
49                 << "\n1" << setw( 13 ) << frequency1
50                 << "\n2" << setw( 13 ) << frequency2
51                 << "\n3" << setw( 13 ) << frequency3
52                 << "\n4" << setw( 13 ) << frequency4
53                 << "\n5" << setw( 13 ) << frequency5
54                 << "\n6" << setw( 13 ) << frequency6 << endl;
55
```

```
56     return 0;
57 }
```

输出结果:

Face	Frequency
1	987
2	984
3	1029
4	974
5	1004
6	1022

图 3.8 模拟掷骰子 6 000 次

如输出结果所示,通过按比例缩放和移动,我们可以利用 rand 函数来实际模拟掷骰子。注意,程序决不会用 switch 结构提供的 default case,但不管怎样,我们仍然将其作为良好的编程习惯来提供。第 4 章介绍数组时,将介绍如何把整个 switch 结构替换为单行语句。

**测试和调试提示 3.1** 即使能完全确定程序中没有错误,也应该在 switch 结构中使用 default 条件来捕捉错误。

再次执行图 3.7 中的程序,将生成

5	5	3	5	5
2	4	2	5	5
5	3	2	2	1
5	1	4	6	4

注意,打印出来的数值顺序与上一次完全相同。这怎么能算得上是随机数呢?具有讽刺意味的是,这种可重复性是 rand 函数的一个重要特性。调试程序时,这种可重复性是证明程序能正常工作的一个关键。

函数 rand 实际上生成的是“伪随机数”(Pseudo-random numbers)。重复调用 rand 会生成看上去是随机产生的一系列数值。然而,这一系列数值会在程序执行时重复出现。一旦程序已经过彻底调试,就可以实际调整为每次执行程序时生成不同的随机数系列。这个过程被称为“随机化”(randomizing),是利用标准库函数 srand 来实现的,函数 srand 采用了一个无符号参数,并内嵌了一个 rand 函数,以便在每次执行程序时生成不同的随机数系列。

srand 的用法如图 3.9 所示。在其中的程序中,我们采用了数据类型 unsigned (unsigned int 的缩写)。int 值至少应占内存的两个字节。两字节的 unsigned int 只能取 0 ~ 65 535 之间的非负值,4 字节的 unsigned int 只能取 0 ~ 4 294 967 295 之间的非负值。srand 函数将 unsigned int 值作为自己的参数。srand 函数的函数原型包含在 <cstdlib> 头文件中。

```
1 //Fig.3.9: fig03_09.cpp
2 //Randomizing die-rolling program
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
```

```

8
9 #include <iomanip>
10
11 using std::setw;
12
13 #include <cstdlib>
14
15 int main()
16 {
17     unsigned seed;
18
19     cout << "Enter seed: ";
20     cin >> seed;
21     srand( seed );
22
23     for ( int i = 1; i <= 10; i++ ) {
24         cout << setw( 10 ) << 1 + rand() % 6;
25
26         if ( i % 5 == 0 )
27             cout << endl;
28     }
29
30     return 0;
31 }

```

输出结果:

Enter seed:67

1	6	5	1	4
5	6	3	1	2

Enter seed:432

4	2	6	4	3
2	5	1	4	4

Enter seed:67

1	6	5	1	4
5	6	3	1	2

图 3.9 随机化掷骰子程序

将该程序运行多次并观察其结果。注意,假如每次提供的“种子值”不同,程序每次运行所生成的随机数系列也会“不同”。

如果希望无需每次输入种子值,随机化程序可能应采用语句

```
srand( time( 0 ) );
```

这会令计算机通过自己的时钟值来自动获得种子值。time 函数(前一个语句中,其参数为 0)返回当前“日历时间”的秒数。这个值被转换为 unsigned 整数,并被用作随机数生成器中的种子。time 的函数原型包含在 <ctime> 头文件中。

**性能提示 3.2** srand 函数只需在程序中调用一次,就可以获得所需的随机化结果。多次调用不仅徒劳无益,还会降低程序性能。

rand 直接生成的数值通常在下一范围内

```
0 ≤ rand() ≤ RAND_MAX
```

取值。我们前面演示了如何写一个简单的语句来模拟掷骰子,如下所示

```
face = 1 + rand() % 6;
```

它往往把一个整数值(随机)指定给取值范围在  $1 \leq \text{rand}() \leq 6$  之间的变量 face。注意,该范围的宽度(也就是范围中的连续整数的个数)是 6,而且范围中的起始数值是 1。从前面的语句可以看出,范围宽度是由带有求模操作符的按比例缩放 rand 函数中的数值类决定的(也就是 6),而且范围的起始数值与添加到  $\text{rand} \% 6$  中的数值(也就是 1)相同。我们对这个结果进行了一般化处理

```
n = a + rand() % b;
```

a 指的是“位移值”(相当于所需的连续整数范围内的宽度值)。练习题中,我们将介绍如何从一组非连续性整数中随机选择整数。

**常见编程错误 3.16** 用 srand 取代 rand,令其生成随机数是语法错误,因为 srand 函数没有返回值。

### 3.9 示例:博彩游戏和 enum 简介

在目前流行的博彩游戏中,有一种名为“掷双骰”的游戏,世界各地都可见到它的踪影。它的规则十分简单:玩家滚动两粒骰子。每个骰子都有六面,分别标上 1,2,3,4,5 和 6 这几个点。两粒骰子停止滚动之后,计算出它们朝上那一面的点数之和。假如首次掷出后点数之和为 7 或 11,那么玩家赢(庄家输)。假如首次掷出后点数之和为 2,3 或者 12,那么玩家输(庄家赢)。假如首次掷出后点数之和为 4,5,6,8,9 或者 10,那么这些数字会立即成为玩家的“目标点”。要想赢,必须不断地抛掷骰子,直到点数同这个目标点相同为止。但在这之前,假如不幸地掷出了 7 点,那么玩家马上会输。

图 3.10 中的程序模拟了掷双骰游戏,图 3.11 展示了几个作为示例的执行结果。

```
1 //Fig.3.10; fig03_10.cpp
2 //Craps
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstdlib>
9
10 #include <ctime>
11
12 using std::time;
13
14 int rollDice( void );    //function prototype
15
16 int main()
17 {
18     enum Status { CONTINUE, WON, LOST };
19     int sum, myPoint;
```

```
20     Status gameStatus;
21
22     srand( time( 0 ) );
23     sum = rollDice();                //first roll of the dice
24
25     switch( sum ) {
26         case 7:
27             case 11:                //win on first roll
28                 gameStatus = WON;
29                 break;
30         case 2:
31         case 3:
32             case 12:                //lose on first roll
33                 gameStatus = LOST;
34                 break;
35         default:                    //remember point
36             gameStatus = CONTINUE;
37             myPoint = sum;
38             cout << "Point is" << myPoint << endl;
39             break;                  //optional
40     }
41
42     while ( gameStatus == CONTINUE ) { //keep rolling
43         sum = rollDice();
44
45         if ( sum == myPoint )        //win by making point
46             gameStatus = WON;
47         else
48             if ( sum == 7 )          //lose by rolling 7
49                 gameStatus = LOST;
50     }
51
52     if ( gameStatus == WON )
53         cout << "Player wins" << endl;
54     else
55         cout << "Player loses" << endl;
56
57     return 0;
58 }
59
60 int rollDice( void )
61 {
62     int die1, die2, workSum;
63
64     die1 = 1 + rand() % 6;
65     die2 = 1 + rand() % 6;
66     workSum = die1 + die2;
67     cout << "Player rolled" << die1 << " + " << die2
68         << " = " << workSum << endl;
69
70     return workSum;
}
```



71 |

图 3.10 模拟掷双骰游戏的程序

输出结果:

```

player rolled 6 + 5 = 11
player wins

player rolled 6 + 5 = 11
player wins

player rolled 4 + 6 = 10
point is 10

player rolled 2 + 4 = 6
player rolled 6 + 5 = 11
player rolled 3 + 3 = 6
player rolled 6 + 4 = 10
player wins

player rolled 1 + 3 = 4
point is 4

player rolled 1 + 4 = 5
player rolled 5 + 4 = 9
player rolled 4 + 6 = 10
player rolled 6 + 3 = 9
player rolled 1 + 2 = 3
player rolled 5 + 2 = 7
player loses

```

图 3.11 掷双骰游戏模拟程序的输出结果

注意,玩家在首次掷骰子时,必须先抛两粒骰子,以后同理。我们定义了一个 rollDice 函数掷骰子计算并打印它们的点数和。函数 rollDice 只定义了一次,但可从程序中两个地方调用它。有趣的是,rollDice 没有采用参数,所以我们在参数表中指明了 void。rollDice 函数的确返回了两粒骰子的总和,所以,int 的返回类型会在函数首部指明。

该游戏相当复杂。第一次掷骰子时,玩家可能赢,可能输,在随后的掷骰子过程中,也可能有输有赢。gameStatus 变量用于跟踪输赢状态。gameStatus 变量被声明为 Status 类型。下一行

```
enum Status { CONTINUE, WON, LOST };
```

创建了一个“用户自定义类型”(User-defined type),名为“枚举”(enumeration)类型。枚举类型采用了关键字 enum 和一个类型名称(这里是 Status),它是一组用标识符来表示的整数常量。这些“枚举常量”(enumeration constants)的取值从 0 开始,增量为 1(除非另行指定)。在前面的枚举中,CONTINUE 被指定为 0,WON 被指定为 1 以及 LOST 被指定为 2。enum 中的标识符必须是惟一的,但不同的枚举常量可取用相同的值。

**良好编程习惯 3.6** 使用用户自定义类型名称的标识符,其首字母应该大写。

用户自定义类型 Status 的变量只能赋给枚举中声明的 3 个数值中的一个。游戏取胜时,gameStatus 会被设置为 WON。当游戏输掉时,gameStatus 会被设置为 LOST。否则 gameStatus 会被设置为 CONTINUE,玩家再次开始掷骰子。

**常见编程错误 3.17** 为枚举类型的变量分配一个等同于枚举变量的整数值是语法错误。

另一种常见的枚举类型

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, AUG,
             SEP, OCT, NOV, DEC};
```

它利用枚举类型代表一年中的每个月,创建了一个用户自定义类型 Months。由于上述枚举中的第一个值被显式设置为 1,所以后面的值均从 1 开始递增,取值范围从 1 到 12。在枚举定义中,任何枚举常量都可以被设置为整数值,而且后续的数值均以 1 为增量递增。

**常见编程错误 3.18** 枚举常量已被定义之后,再次尝试为其分配另一个值是语法错误。

**良好编程习惯 3.7** 在枚举常量中,只采用大写字母。这样可以使这些常量在程序中更能引起程序员注意,而且,还可以提醒程序员枚举常量并不是变量。

**良好编程习惯 3.8** 在程序中,使用枚举来代替整数常量可以使程序更清晰。

首次掷骰子之后,如果游戏获胜,就跳过 while 结构主体,因为 gameStatus 并不等于 CONTINUE。程序继续到 if/else 结构,如果 gameStatus 等于 WON,该结构将打印出“Player wins”;如果 gameStatus 等于 LOST,该结构将打印出“Player loses”。

首次掷骰子之后,如果游戏没有结束,sum 就会被保存在 myPoint 中。程序执行权继续到 while 结构,因为 gameStatus 等于 CONTINUE。每次执行 while 结构中的程序时,都会调用 rollDice,生成一个新的 sum。如果 sum 与 myPoint 匹配,gameStatus 就会被设置为 WON,如果 while 测试失败,if/else 结构就会打印出“Player wins”,并中止程序的执行。如果 sum 等于 7,gameStatus 被设置为 LOST,while 测试就会失败,if/else 语句将打印出“Player loses”,并中止程序的执行。

注意我们已经讨论过的不同程序控制机制的有趣使用。掷骰子游戏采用了两个函数(main 和 rollDice),switch,while,if/else 和嵌套式 if 结构。在随后的练习题中,我们将深入探讨掷骰子游戏的各种有趣特性。

## 3.10 存储类

第 1~3 章使用了标识符作为变量名。变量属性中包含了名称、类型、长度和数值。本章,我们还要将标识符用作用户自定义函数名。事实上,程序中的每个标识符还有其他属性,其中包括“存储类”、“作用域”和“链接”。

C++ 提供了 5 个存储类说明符:auto,register,extern,mutable 和 static。一个标识符的存储类说明符有助于帮助你判断该标识符的存储类、作用域和连接。本节将讨论存储类说明符 auto,register,extern,mutable 和 static。存储类说明符 mutable(将在第 21 章详细讨论)随同名为“类”的 C++ 用户自定义类(将在第 6 章和第 7 章详细讨论)一起被广泛使用。

标识符的“存储类”决定了标识符在内存中存在的期限。有的标识符存在的时间较短,而有的标识符可以重复创建和删除,有的标识符则存在于整个程序执行期间。本节,我们要讨论两个存储类:静态存储类(static)和自动存储类(automatic)。

标识符的“作用域”指的是可以从程序中引用标识符的区域。有的标识符可以在整个程

序中引用,有的则只能限于在程序的某个部分引用。3.11 节将讨论标识符的作用域。

对于多源文件程序(我们将在第6章深入讨论)来说,标识符的“链接”将决定是在当前源文件还是在带有正确声明的任何为源文件中识别标识符。

存储类说明符可以分为两个存储类:“自动存储类”和“静态存储类”。关键字 `auto` 和 `register` 用于声明自动存储类的变量。此类变量是在进入声明的块时创建的,它们只存在与块被激活的期间,当程序执行退出块时,这些变量就会被删除。

只有变量可以作为自动存储类,函数的局部变量和参数通常属于自动存储类。存储类说明符 `auto` 显式声明了自动存储类的变量。例如,下面的声明指出 `double` 类型的变量 `x` 和 `y` 是自动存储类的局部变量,也就是说,它们只存在于出现函数定义的函数主体

```
auto double x, y;
```

局部变量被默认为自动存储类,所以关键字 `auto` 很少使用。在本节随后的描述中,我们将把自动存储类变量键称为自动变量。

**性能提示 3.3** 自动存储意味着可以节省内存,因为自动存储类变量会在进入声明的块时创建,并在退出块时被删除。

**软件工程知识 3.11** 自动存储是最低权限级的实例。在不需要变量时,为何要将其保存在内存中供存取呢?

机器语言版本中的数据通常被装入寄存器(register),供计算和其他处理使用。

**性能提示 3.4** 存储类说明符 `register` 可以置于自动变量声明之前,以令编译器在计算机的高速硬件寄存器中而不是在内存中维护该变量。如果可以在硬件寄存器中维护频繁使用的变量(比如计数器、总和),那么重复将变量从内存装入寄存器以及将结果返回内存而引起的开销就可以避免。

**常见编程错误 3.19** 同一个标识符使用多个存储类说明符是语法错误。一个标识符只能使用一个存储类说明符。例如,如果把一个标识符指定为 `register`,就不能再将其设为 `auto`。

编译器可能会忽略 `register` 声明。例如,编译器可用的寄存器数目可能不够。声明

```
register int counter = 1;
```

“建议”你把变量 `counter` 置于编译器的寄存器中;不管编译器是否这样做, `counter` 都会被初始化为1。关键字 `register` 只能随局部变量和函数参数一起使用。

**性能提示 3.5** 通常情况下, `register` 声明并非必不可少。如今的优化编译器能识别使用较为频繁的变量,并能在程序员不提供 `register` 声明的情况下,将这些变量置于寄存器中。

关键字 `extern` 和 `static` 是用于声明变量和静态存储类函数的说明符。此类变量存在于程序开始执行时。对变量而言,程序开始执行时就为其分配和初始化存储空间。对于函数而言,函数名称将从程序开始执行时就得以存在。但是,即使变量和函数名会从程序开始执行时就存在,并不意味着这些标识符可应用于整个程序中。存储类和作用域(可以使用名称的地方)是两个不同的概念,我们将在3.11节详细讨论。

静态存储类有两类标识符:外部标识符(比如全局变量和函数名称)和随同存储类说明符 `static` 一起声明的局部变量。全局变量和函数名被默认为存储类说明符 `extern`。至于全局

变量的创建,把变量声明放于任何函数定义之外即可。全局变量的值在程序执行的整个过程中保持不变。全局变量和函数可以被任何一个函数引用,只要源文件中已该函数的其声明或定义。

**软件工程知识 3.12** 如果把变量声明为全局变量而非局部变量,那么对于不需要访问变量的函数而言,如果它有意或无意修改该变量时,可能引起难以预料的副作用。一般说来,除非有特殊的性能需求,多数情况下都应尽量避免使用全局变量。

**软件工程知识 3.13** 只用于特定函数的变量应该声明为该函数的局部变量而非全局变量。

利用关键字 `static` 声明的局部变量仍然可以为定义它们的函数识别,但不同于自动变量,`static` 局部变量会在程序退出函数时仍然保留其值。下一次调用该函数时,`static` 局部变量将在程序最后退出该函数时保留其值。语句

```
static int count = 1;
```

把局部变量 `count` 声明为 `static`,并将其初始化为 1;如果程序员没有显式初始化于静态存储类的数字变量,那么所有数字变量都会被初始化为 0(将在第 5 章讨论的静态指针变量也会被初始化为 0)。

存储类说明符 `extern` 和 `static` 在被显式应用于外部标识符时,有特殊含义。第 18 章将讨论 `extern` 和 `static` 应用于外部标识符和多源文件程序时的用法。

### 3.11 作用域规则

程序中,标识符有含义的部分被称为它的“作用域”。例如,我们块中声明一个局部变量时,它只能在在该块中或这在该块内嵌套的另一个块中使用。标识符有 4 个作用域,分别是:函数范围、文件范围、块范围和函数原型范围。稍后我们还要介绍另外两个作用域——类范围(第 6 章)和名称空间范围(第 21 章)。

任何函数外部定义的标识符都有“文件范围”。此类标识符可以供声明处直到文件结束处的所有函数访问。位于函数外部的全局变量、函数定义和函数原型都有自己的文件范围。

“标号”即后面带有一个冒号的标识符(比如 `start:`)是惟一具有“函数范围”的标识符。标号可用于所在函数的任何位置,但不能在函数主体外部进行引用。标号还可用于 `switch` 结构(比如 `case` 标号)和 `goto` 语句(参见第 18 章)。标号隐藏于函数内部的实现细节。隐藏——更通俗的名称是信息隐藏——是优秀软件工程应具有的最基本的原则。

块内部声明的标识符有“块范围”。块范围始于标识符的声明处,结束于块的右花括号(`}`)。函数开始处声明的局部变量有块范围,作为函数局部变量的函数参数也如此。任何一个块都包含变量声明。块被嵌入,且外部块中的标识符与其内部块中的标识符同名时,外部块中的标识符会处于隐藏状态,直到内部块中止执行为止。程序在内部块中执行时,内部块中的标识符值是在本块中定义的,而不是同名的外部标识符值。声明为 `static` 的局部变量也有块范围,即使它们只存在于程序开始执行时。存储期限不会对标识符的作用域产生影响。

惟一具有“函数原型范围”的标识符是函数原型参数列表中所用的标识符。如前所述,

函数原型不需要参数列表中的名称——只需要其类型。如果使用了函数原型参数列表中的名称,编译器也会将其忽略。函数原型中使用的标识符可以重复用于程序其他地方,而不会引起歧义。

**常见编程错误 3.20** 如果出于偶然,对内部块和外部块中的标识符采用了同样的名称,而程序员事实上却希望外部块中的标识符在内部块存在期间处于活动状态,通常属于逻辑错误。

**良好编程习惯 3.9** 避免隐藏外部块范围的变量名。这是通过避免在程序中使用重复性的标识符来实现的。

图 3.12 中的程序演示了全局变量、自动局部变量和静态局部变量的作用域问题。

```

1 //Fig. 3.12: fig03_12.cpp
2 //A scoping example
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void a( void ); //function prototype
9 void b( void ); //function prototype
10 void c( void ); //function prototype
11
12 int x = 1; //global variable
13
14 int main()
15 {
16     int x = 5; //local variable to main
17
18     cout << "local x in outer scope of main is " << x << endl;
19
20     | //start new scope
21     int x = 7;
22
23     cout << "local x in inner scope of main is " << x << endl;
24     | //end new scope
25
26     cout << "local x in outer scope of main is " << x << endl;
27
28     a(); //a has automatic local x
29     b(); //b has static local x
30     c(); //c uses global x
31     a(); //a reinitializes automatic local x
32     b(); //static local x retains its previous value
33     c(); //global x also retains its value
34
35     cout << "local x in main is " << x << endl;
36
37     return 0;

```

```

38 {
39
40 void a( void )
41 {
42     int x = 25; //initialized each time a is called
43
44     cout << endl << "local x in a is " << x
45         << " after entering a" << endl;
46     ++x;
47     cout << "local x in a is " << x
48         << " before exiting a" << endl;
49 }
50
51 void b( void )
52 {
53     static int x = 50;    //Static initialization only
54                          //first time b is called.
55     cout << endl << "local static x is " << x
56         << " on entering b" << endl;
57     ++x;
58     cout << "local static x is " << x
59         << " on exiting b" << endl;
60 }
61
62 void c( void )
63 {
64     cout << endl << "global x is " << x
65         << " on entering c" << endl;
66     x *= 10;
67     cout << "global x is " << x << " on exiting c" << endl;
68 }

```

输出结果:

```

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

```

```

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c

```

```
global x is 100 on exiting c
local x in main is 5
```

图 3.12 作用域示例

全局变量  $x$  被声明并初始化为 1。该全局变量在其中声明了  $x$  变量的任何一个块(或函数)中,都是隐藏的。在 `main` 中,局部变量  $x$  被声明并初始化为 5。该变量被打印出来,说明全局变量  $x$  在 `main` 中是隐藏的。接下来,`main` 中定义了一个新块,其中另一个局部变量  $x$  被初始化为 7。该变量也被打印出来,说明它在 `main` 的外部块中隐藏了  $x$ 。值为 7 的变量  $x$  会在程序退出块时自动删除,`main` 外部块中的局部变量  $x$  被打印出来,表示其不再隐藏。示例程序定义了 3 个函数——每个函数均没有参数和返回值。函数 `a` 定义了自动存储类变量  $x$ ,并将其初始化为 25。程序调用 `a` 时,将打印该变量,递增其值,并在退出该函数之前,再次打印该变量。每次调用该函数时,自动变量  $x$  会被重新创建并初始化为 25。函数 `b` 声明了静态变量  $x$ ,并将其初始化为 50。声明为 `static` 的局部变量即使不在作用域内,也会保留其值。程序调用 `b` 时,将打印  $x$  变量,递增其值,并在退出函数之前,再次打印该变量。函数 `c` 没有声明任何变量。因此,当它引用变量  $x$  时,会打印全局变量,将其乘以 10,并在退出函数时,再次打印变量。下一次调用函数 `c` 时,全局变量仍然会保留其修改过的值 10。最后,程序再次打印 `main` 中的局部变量  $x$ ,表示所有函数调用都没有改变  $x$  的值,因为函数引用的都是其他作用域中的变量。

## 3.12 递归

我们介绍的程序通常由调用另一个函数的函数组成,这些函数具有严格的层次结构。但为了解决某些问题,令函数调用其本身将非常有用。“递归函数”是可以通过另一个函数调用其自身的函数——这种调用既可以是直接进行的,也可以是间接进行的。“递归”是高级计算机课程的一个重点主题。本节以及下一节,我们将介绍一些简单的递归例子。本书则涵括了大量的递归处理例子。

我们先讨论一下递归的概念,然后再介绍几个其中包含递归函数的实例。各种递归问题都有一个共通的解决办法。当我们调用递归函数,打算用来解决问题时,函数实际只知道如何应付最简单的情况——或称“基本条件”。假如在一个基本条件下调用函数,函数只是简单地返回一个结果。假如在比较复杂的情况下调用函数,那么函数会把问题分割成两个概念性的部分——一部分函数知道该怎么做,另一部分函数则不知道该怎么做。为使递归变得可行,后一部分必须在表面上“像”原始问题,但要显得稍微简单一些,或稍微小一些。由于这个新问题看起来和原来的问题颇为相似,所以函数能启动自己的一个新副本(也就是调用自己),对较小的问题进行处理——这便是所谓的“递归调用”,也称作“递归处理”。递归调用时,通常要用到关键词 `return`,这是由于它的结果将同已知如何解决的那一部分问题合并到一起,以便将结果返回原调用者——可能是 `main`。

原来对函数的调用仍处在“打开”状态——也就是上一次执行尚未完成期间,递归步骤会得以执行。递归步骤会造成更多的递归调用,因为函数会持续把函数调用的每个新的子问题分解为两个概念性的部分。为了最终中止递归,每次函数调用自身时,都会被分为越来越

越细的问题,最终将“递归”合并为基本条件。此时,函数可识别出基本条件,并把结果返回前一个函数副本,并“回归”到一系列结果,直到原函数最终把最后的结果返回 main。所有这些似乎都比前面介绍的问题复杂。为进一步说明这些概念的用法,我们要编写一个递归程序,用于执行一个常见的数学运算。

对非负整数  $n$  来说,它的阶乘写作  $n!$  (念作“ $n$  的阶乘”),公式如下:

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

$1!$  等于 1,而  $0!$  也被定义成 1。例如,  $5!$  等于  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ ,结果等于 120。

对于大于或等于 0 的整数 number 来说,它的阶乘可以利用 for 循环迭代(非递归方式)来计算,如下所示

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
    factorial *= _counter;
```

观察下述关系,我们可总结出阶乘的一个递归定义:

$$n! = n \cdot (n-1)!$$

例如,  $5!$  显然等于  $5 \cdot 4!$ ,如下所示

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

$5!$  计算的结果如图 3.13 所示。图 3.13a)展示了如何递归调用,直到  $1!$  的结果为 1,此时程序会中止递归。图 3.13b)显示了每次递归调用向其调用者返回的值,直到计算并返回最终值。

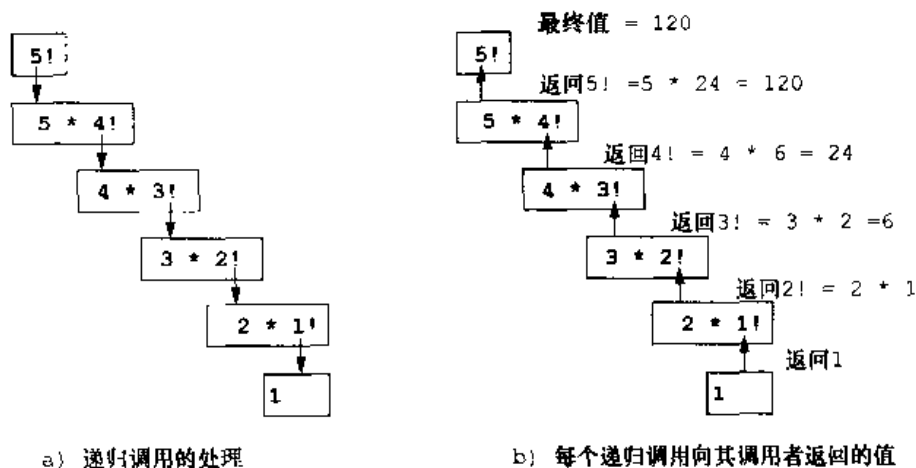


图 3.13 计算  $5!$  的结果

图 3.14 中的程序用递归来计算并打印 0 到 10 之间的整数(unsigned long 数据类型的选择会在稍后进行讨论)的阶乘。递归函数 factorial 首先测试中止条件是否为 true,也就是 number 是否小于或等于 1。如果 number 的确小于或等于 1, factorial 会返回 1,不再需要继续递归,程序将中止。如果 number 大于 1,语句

```
return number * factorial( number - 1 );
```



将把问题表示为 number 乘以 factorial 递归函数计算的 number - 1 的阶乘。注意, factorial (number - 1) 比原来的计算 factorial (number) 更简单。

```

1 //Fig.3.14: fig03_14.cpp
2 //Recursive factorial function
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 unsigned long factorial( unsigned long );
13
14 int main()
15 |
16     for ( int i = 0; i <= 10; i ++ )
17         cout << setw( 2 ) << i << "!=" << factorial( i ) << endl;
18
19     return 0;
20 |
21
22 //Recursive definition of function factorial
23 unsigned long factorial( unsigned long number )
24 |
25     if ( number <= 1 ) //base case
26         return 1;
27     else //recursive case
28         return number * factorial( number - 1 );
29 |

```

输出结果:

```

0! =1
1! =1
2! =2
3! =6
4! =24
5! =120
6! =720
7! =5040
8! =40320
9! =362880
10! =3628800

```

图 3.14 用递归函数计算阶乘

函数 factorial 被声明为接收 unsigned long 类型的参数,并返回 unsigned long 类型的结果值。unsigned long 是 unsigned long int 的缩写。C++ 语言规范中,要求 unsigned long int 类型的变量至少占 4 个字节(32 位),因此它的取值范围可以在 0 ~ 4 294 967 295 之间(long int 数据类型至少占 4 个字节,取值范围在正负 2 147 483 647 之间)。如图 3.14 所示,阶乘值迅

速变得很大。我们已选定了 unsigned long 数据类型,所以程序可以在字长较小(比如 2 字节)的计算机上计算大于  $7i$  的阶乘。不幸的是, factorial 函数如此迅速地产生了较大的值,以至于在超出 unsigned long 变量的长度之前, unsigned long 也不能帮助我们计算多个阶乘值。

我们将在练习题中谈到,希望计算大型数目阶乘的用户最终会使用 double 类型。这便指出了许多编程语言的不足,也就是说语言不能轻松得以扩展,以处理不同应用程序的恶独特需求。我们将在本书的面向对象编程部分介绍, C++ 是一个扩展性很强的语言,它可以让你在需要的时候,创建任意大的数。

**常见编程错误 3.21** 需要递归函数返回值时,如果忘记返回值将导致大多数编译器生成一条警告消息。

**常见编程错误 3.22** 省略基本条件,或把递归步骤误写为不能到推回基本条件都可能导致“无穷递归”,最终耗尽内存。这如同在迭代(非递归)解决方法中的无限循环问题。导致无穷递归的另一种可能是意外的输入错误。

### 3.13 递归应用示例:费波拉奇数列

费波拉奇(Fibonacci)数列

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

肯定从 0 和 1 开始,后续每个费波拉奇数字都是前两个数字的和。

自然界中存在这种数列,它描述了一种“螺旋分割”形式。相邻的费波拉奇数之比接近于一个固定值 1.618...。这个数在自然界中经常出现,被称作“黄金比”或者“黄金分割比”。人类在很早的时候就发现,黄金分割的物体往往具有更强烈的美感。因此,建筑师通常按这一比例来设计窗户、房间等等,使它们的长宽比正好为黄金分割比。另外,邮政明信片的长与宽往往也按此比率设计。

费波拉奇数列可像下面这样递归定义

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

图 3.15 中的程序用 fibonacci 函数递归计算第  $i$  个费波拉奇数。注意,费波拉奇数会很快变大。因此,我们把 fibonacci 函数中的参数类型和返回类型均设为 unsigned long。图 3.15 中,每对输出行都展示了单独的运行结果。

```
1 //Fig. 3.15: fig03_15.cpp
2 //Recursive fibonacci function
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 unsigned long fibonacci( unsigned long );
```

```
10
11 int main()
12 {
13     unsigned long result, number;
14
15     cout << "Enter an integer; ";
16     cin >> number;
17     result = fibonacci( number );
18     cout << "Fibonacci(" << number << ") = " << result << endl;
19     return 0;
20 }
21
22 //Recursive definition of function fibonacci
23 unsigned long fibonacci( unsigned long n )
24 {
25     if ( n == 0 || n == 1 )    //base case
26         return n;
27     else                      //recursive case
28         return fibonacci( n - 1 ) + fibonacci( n - 2 );
29 }
```

输出结果:

```
Enter an integer:0
Fibonacci(0) = 0

Enter an integer:1
Fibonacci(1) = 1

Enter an integer:2
Fibonacci(2) = 1

Enter an integer:3
Fibonacci(3) = 2

Enter an integer:4
Fibonacci(4) = 3

Enter an integer:5
Fibonacci(5) = 5

Enter an integer:6
Fibonacci(6) = 8

Enter an integer:10
Fibonacci(10) = 55

Enter an integer:20
Fibonacci(20) = 6765

Enter an integer:30
Fibonacci(30) = 832040

Enter an integer:35
Fibonacci(35) = 9227465
```

图 3.15 递归生成费波拉奇数列

从 main 中调用 fibonacci 并不是递归调用,但随后的 fibonacci 调用都是递归调用。每次

调用 fibonacci 函数时,该函数都会立即测试基本条件—— $n$  等于 0 还是 1。如果测试结果为真,就返回  $n$ 。有趣的是,如果  $n$  大于 1,递归步骤就会产生两个递归调用,每个调用都比对 fibonacci 的第一次调用简单。图 3.16 展示了 fibonacci 函数如何计算 fibonacci(3)——为使程序图更清晰,我们将把 fibonacci 简写为  $f$ 。

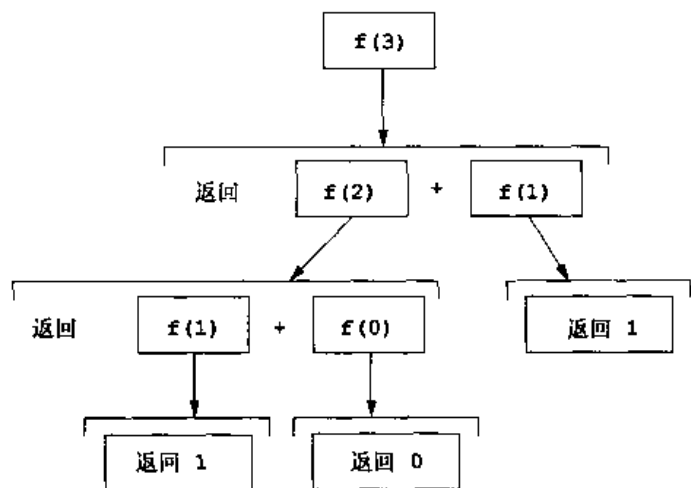


图 3.16 fibonacci 函数的一系列递归调用

该图提出了几个有趣的问题,这些问题与 C++ 编译器计算操作符操作数的顺序有关。这种顺序不同于操作符应用于其操作数的顺序,后者的顺序由操作符优先级来确定的。从图 3.16 中可以看出,计算  $f(3)$  期间,会执行两个递归调用,分别为  $f(2)$  和  $f(1)$ 。但这些调用应采用哪种顺序呢?

绝大多数程序员会武断认为,操作数采用从左到右的顺序求值。奇怪的是, C++ 语言没有指定大多数操作符(包括  $+$  号在内)的操作数求值顺序。因此,程序员必须杜绝假设这些调用的执行顺序。事实上,调用可能先执行  $f(2)$  然后执行  $f(1)$ ,也可能先执行后者再执行前者。在这个程序和大多数其他程序中,最后的结果都是一样的。但在某些程序中,操作数求值顺序会有负面影响,会影响表达式的最终结果。

C++ 语言只指定了 4 个操作符的操作数求值顺序。这 4 个操作符分别是:  $\&\&$ 、 $||$ 、 $,$  和  $?:$ 。前 3 个是二进制操作符,其两个操作数的求值顺序是从左到右。最后一个操作符是 C++ 语言中惟一的三元操作符。求值时,通常先对其最左边的操作数求值。如果最左边的操作数为非 0,就对种间的操作数求值,忽略最后一个操作数;如果最左边的操作数为 0,就对第 3 个操作数求值,忽略中间那个操作数。

**常见编程错误 3.23** 对于非  $\&\&$ 、 $||$ 、逗号  $,$  和  $?:$  操作符的操作符,如果要想写依赖于其求值顺序的程序,可能会出现错误,因为编译器也许不会按程序员期望的那样对操作数进行求值。

**可移植性提示 3.2** 依赖于非  $\&\&$ 、 $||$ 、逗号  $,$  和  $?:$  操作符操作数的程序在系统中的作用会因为编译器的不同而不同。

要注意,类似的递归程序会产生费波拉奇数列。函数 fibonacci 中的每个递归级都会使调用数增加一倍,也就是说第  $n$  个费波拉奇数的求值将在第  $2^n$  次递归调用时进行。这样产

生的数值很快就难以控制。对第 20 个费波拉奇数的求值就需要  $2^{20}$  次,也就是需要上百万次。对第 30 个费波拉奇数的求值则需要  $2^{30}$  次,也就是上十亿次,依此类推。计算机科学家将这种现象称为“指数复杂性”。这个问题甚至能让世界上最快的计算机难以胜任!名为“算法”的高级计算机科学课程会介绍的—般的复杂性和特殊情况下的指数复杂性问题。

**性能提示 3.6** 避免使用会造成调用呈指数级递增的费波拉奇式递归程序。

## 3.14 递归和迭代的对比

前面,我们学习了两个可以通过递归和迭代方式轻松实现的函数。本节,我们要对这两个函数进行比较,并讨论程序员在不同情况下应选择的方法。

递归和迭代方式均基于一个控制结构:迭代使用的是重复结构,递归使用的则是选择结构。迭代和递归均涉及到重复;迭代显式使用一个重复结构,递归则通过重复性的函数调用来实现重复。迭代和递归均涉及到中止测试:迭代会在循环条件失败时中止,递归则是在碰到基本条件时中止。迭代会不断控制着计数器,递归则随抵达中止点时逐步重复;迭代会不断修改计数器,直到碰到令循环条件失败的计数器值;递归则不断产生原始问题的简化副本,直到碰到基本条件为止。迭代和递归均无限;如果循环条件测试不可能为假,迭代就会产生无限循环;如果递归步骤不能回归到基本条件,就会发生无穷递归。

递归有很多负面影响。它采用重复调用函数的机制,不断产生着开销。这会浪费处理器处理时间,还会占用内存。每个递归调用都会产生另一个函数副本(事实上产生的只是函数变量);这样会占用大量内存空间。迭代通常在函数内部发生,所以不存在重复调用函数和额外的内存分配这方面的开销。那么我们为什么仍然要选择递归呢?

**软件工程知识 3.14** 可以采用递归方式处理的任何问题也可采用迭代方式(非递归方式)解决。递归法更能反映问题并令程序易于理解和调试时,递归法通常是优于迭代法。选择递归法的另一个理由是迭代法不直观。

**性能提示 3.7** 避免在对性能要求较高的情形下使用递归。递归调用既费时又占用较多的内存。

**常见编程错误 3.24** 出于失误,令非递归函数通过另一个函数直接或间接调用其本身是逻辑错误。

和我们的做法不同,绝大部分程序设计书都把递归放在后面的章节介绍。我们感到递归是一个具有丰富内涵而又复杂的主题,应该早点介绍,并在本书的其他地方列出部分示例。图 3.17 对本书中的递归示例和练习进行了总结。

现在,让我们重新考虑本书中不断强调的某些重要观点。优秀的软件工程固然重要,但高性能同样不可缺少。不幸的是,这些目标犹如鱼翅与熊掌,不可兼而得之。优秀的软件工程是令大型复杂软件系统开发任务易于管理的重要环节。这些系统的高性能也是日后在增加硬件计算需求时,实现这些系统的关键。这两个关键应该如何取得平衡呢?

章	递归示例和练习
第3章	阶乘函数 Fibonacci 函数 最大公约数 两个整数之和 两个整数之积 一个整数的整数次幂 汉诺塔 逆向打印键盘输入 可视化递归
第4章	数组元素的求和 打印数组 逆向打印数组 逆向打印字符串 检查字符串是否为回文 数组中的最小值 选择性排序 八皇后 线性搜索 二元搜索
第5章	快速排序 走迷宫 逆向打印键盘输入的字符串
第15章	链表的插入 链表的删除 链表的查找 逆向打印链表 二叉树的插入 二叉树的前序遍历 二叉树的中序遍历 二叉树的后序遍历

图 3.17 本书的递归示例和练习

**软件工程知识 3.15** 令程序以清晰的层次化结构进行运行,可提高软件工程质量,但要付出一定的代价。

**性能提示 3.8** 一个由多个函数组成的程序——与没有任何函数的一体式程序相比——会产生大量的函数调用,而且这些调用会占用执行时间和计算机处理器的空间。但一体式程序的编程、测试、调试、维护和改进都比较复杂。

一定要使自己编写的程序具有丰富功能,始终牢牢把握性能和优秀软件工程之间的平衡。

### 3.15 使用空参数列表的函数

在C++中,空参数列表的指定是通过 `void` 来指定,或在括号中不写入任何参数来指定的。函数原型

```
void print();
```

指定了函数 `print` 没有任何参数,也不返回值。图 3.18 演示了C++中如何声明和使用不取任何参数的函数。

```
1 //Fig.3.18: fig03_18.cpp
2 //Functions that take no arguments
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void function1();
9 void function2( void );
10
11 int main()
12 {
13     function1();
14     function2();
15
16     return 0;
17 }
18
19 void function1()
20 {
21     cout << "function1 takes no arguments" << endl;
22 }
23
24 void function2( void )
25 {
26     cout << "function2 also takes no arguments" << endl;
27 }
```

输出结果:

```
function1 takes no arguments
function2 also takes no arguments
```

图 3.18 声明和使用无参数函数的两种方式

**良好编程习惯 3.10** 始终记得提供函数原型,即使函数在使用之前的定义时可能被省略。提供函数原型可以避免代码按函数定义的顺序出现(可轻易随程序的改变而改变)。

**可移植性提示 3.3** C++中,空函数参数列表的含义与C中有显著差别。在C语言中,它意味着所有参数检查都是无效的(也就是说,函数调用可以传递任何它想传递的参数)。在C++语言中,则意味着函数没有参数。因此,在C++中编译采用该特性的C程序时,可能会报告语法错误。

既然知道了我们介绍的省略问题,就要注意在调用函数之前,定义在文件中的函数不需要单独的函数原型。此时,函数首部就可以充当函数原型。

**常见编程错误 3.25** C++ 程序不会得以编译,除非提供每个函数的函数原型或在调用函数之前定义每个函数。

## 3.16 内联函数

从软件工程的角度看,把程序作为一个函数集合来实现的确不错,但函数调用会产生运行时开销。C++ 提供了“内联函数”(inline functions)以减少函数调用所产生的开销——尤其对于小型函数。函数定义中,函数返回类型之前的限定符 inline“建议”编译器在适当的程序部分生成函数的副本,以避免函数调用。这样做的结果是程序中插入多个函数代码的副本(因而令程序增大),而不是一个函数的副本(每次调用函数时,控制都会被传递到函数中)。编译器可以忽略 inline 限定符,典型情况下均如此,但小型函数除外。

**软件工程知识 3.16** 对内联函数的任何更改都需要重新编译该函数的所有客户。这会大大影响某些程序的开发和维护。

**良好编程习惯 3.11** inline 限定符应该只适用于小型的、使用较为频繁的函数。

**性能提示 3.9** 使用内联函数会减少执行时间,但会增大程序的长度。

图 3.19 利用内联函数 cube 来计算边长为 s 的立方体的体积。cube 函数的参数列表中,关键字 const 指明编译器函数不能修改变量 s。这样以来便保证了 s 的值不会在执行计算的过程中被函数改变。有关关键字 const 的详情参见第 4 章、第 5 章和第 7 章。

```

1 //Fig.3.19: fig03_19.cpp
2 //Using an inline function to calculate
3 //the volume of a cube.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 inline double cube( const double s ) { return s * s * s; }
11
12 int main()
13 {
14     cout << "Enter the side length of your cube: ";
15
16     double side;
17
18     cin >> side;
19     cout << "Volume of cube with side "
20         << side << " is " << cube( side ) << endl;
21

```



```

22     return 0;
23 |

```

输出结果:

```

Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875

```

图 3.19 利用内联函数计算立方体的体积

**软件工程知识 3.17** 许多程序员不会多此一举地把数值参数声明为常量,即使被调用的函数并不会修改所传递的参数(实参)。使用关键字 `const` 的目的只是保护原始实参的副本,而不是原始实参本身。

### 3.17 引用和引用参数

在许多编程语言中,调用函数的方式有两种,分别是“传值调用”(call-by-value)和“引用调用”(call-by-reference)。参数传值调用时,会产生该参数值的副本并将该副本传递给被调用的函数。对副本的更改不会影响调用者的原始变量值。这样可以避免意外的“负面影响”,从而阻碍正确而稳定的软件系统的开发。迄今为止,本章介绍的程序中,所传递的每个实参都是传值调用的。

**性能提示 3.10** 传值调用的缺点是,如果传递的是大型数据项目,那么复制该数据会花费较长的时间。

本节要介绍“引用参数”(reference parameters)——C++ 为执行引用调用而提供的两种方式之一。利用引用调用,调用者可以令被调用的函数直接访问调用者的数据,并在被调用函数选中该数据的情况下,修改该数据。

**性能提示 3.11** 引用调用对性能有一定好处,因为它可以避免复制大量数据的开销。

**软件工程知识 3.18** 引用调用的安全性较差,因为被调用函数会直接访问并修改调用者的数据。

我们将向大家展示如何在获得调用者数据不受被调用函数干扰的以符合软件工程需要的同时,又能获得引用调用带来的性能优势。

引用参数是其对应形参的别名。要想指明函数参数是按引用进行传递的,只需在函数原型的参数类型后加上一个 `and(&)` 标记;函数头部列出参数类型时,需使用同样的约定。例如,函数头部的声明

```
int &count
```

可能会被认为“count 是对一个 int 类型值的引用。”。在函数调用中,简单地通过名称来指代变量,那么该变量将按引用传递。然后,用其在被调用函数体中的参数名指代该变量,实际上引用的是调用函数中的原始变量,被调用函数可以直接修改原始变量。通常情况下,函数原型和函数头部必须保持一致。

图 3.20 对传值调用、引用调用与引用参数进行了比较。在对 `squareByValue` 和 `squareByReference` 的调用中,所取参数的“形式”是一样的,也就是说,两个变量都是简单地通过名

称来指定的。不检查函数原型或函数定义,就不可能判断出哪个函数修改了自己的参数。因为函数原型是强制性的,编译器可以轻而易举地解决歧义性这一问题。

```

1 //Fig. 3.20: fig03_20.cpp
2 //Comparing call-by-value and call-by-reference
3 //with references.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 int squareByValue( int );
10 void squareByReference( int & );
11
12 int main()
13 {
14     int x = 2, z = 4;
15
16     cout << "x = " << x << " before squareByValue\n"
17         << "Value returned by squareByValue: "
18         << squareByValue( x ) << endl
19         << "x = " << x << " after squareByValue\n" << endl;
20
21     cout << "z = " << z << "before squareByReference" << endl;
22     squareByReference( z );
23     cout << "z = " << z << " after squareByReference" << endl;
24
25     return 0;
26 }
27
28 int squareByValue( int a )
29 {
30     return a * a; //caller's argument not modified
31 }
32
33 void squareByReference( int &cRef )
34 {
35     cRef * = cRef; //caller's argument modified
36 }

```

输出结果:

```

x=2 befor squareByValue
Value returned by squareByValue:4
x=2 after squareByValue
z =4 before squareByReference
z =16 after squareByReference

```

图 3.20 引用调用示例

**常见编程错误 3.26** 由于在被调用函数体中,引用参数只能通过名称来指定,所以程序员可能会因此而把引用参数视为传值调用参数。这样一来,如果变量的原始副本已被调用函

数修改,就会导致不可预测的副作用。

第5章将讨论指针;同时还会讲到指针会提供另一种形式的引用调用,这种形式中,调用形式很清晰地指明了引用调用(以及修改调用者参数的可能性)。

**性能提示 3.12** 对于传递大型对象,可用常量引用参数来模拟传值调用的外观和安全性,进而避免传递大型对象副本的开销。

要想指定常量引用,应在参数声明的类型说明符前添加一个 `const` 限定符。

注意,函数 `squareByReference` 的参数列表中,有一个 `and` 记号(`&`)。而对于使用 `int &cRef` 与 `int &cRef`,有的C++程序员更偏爱前者。

**软件工程知识 3.19** 综合考虑程序的清晰性和高性能,许多C++程序员偏向于使用指针的使用,把可修改的参数传递给函数,小型的非修改性参数可以传值调用,大型的非修改参数则可以利用常量引用传递给函数。

引用还可用作函数内其他变量的别名。例如,代码

```
int count = 1           //declare integer variable count
int &cRef = count;      //creat cRef as an alias for count
++cRef;                 //increment count (using its alias)
```

通过其别名 `cRef` 的使用,递增了变量 `count` 的值。引用变量必须在其声明中得以初始化(参见图 3.21 和图 3.22),而且不能作为其他变量的别名赋予别的值。一旦引用被声明为另一个变量的别名,对该别名执行的所有操作(也就是引用)实际上是在原始变量本身进行的。别名仅是其原始变量的一个名称而已。获取引用的地址和比较引用不会导致语法错误,各项操作实际上是在别名所对应的原始变量上进行的。引用参数必须为左值,而不能是常量或返回左值的表达式。

```
1 //Fig. 3.21: fig03_21.cpp
2 //References must be initialized
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int x = 3, &y = x; //y is now an alias for x
11
12     cout << "x = " << x << endl << "y = " << y << endl;
13     y = 7;
14     cout << "x = " << x << endl << "y = " << y << endl;
15
16     return 0;
17 }
```

输出结果:

x = 3

```

y = 3
x = 7
y = 7

```

图 3.21 使用初始化引用

```

1 //Fig.3.22; fig03_22.cpp
2 //References must be initialized
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int x = 3, &y; //Error: y must be initialized
11
12     cout << "x = " << x << endl << "y = " << y << endl;
13     y = 7;
14     cout << "x = " << x << endl << "y = " << y << endl;
15
16     return 0;
17 }

```

Borland C++ 命令行编译器输出的错误消息:

```
Error E2304 Fig03_22.cpp 10;Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ 编译器输出的错误信息:

```
Fig 03_22.cpp(10):error c2530: 'y';references must be initialized
```

图 3.22 打算使用未经初始化的引用

**常见编程错误 3.27** 在用圆点隔开的变量名称列表中,多处使用 & 的同时,要在语句中声明多个引用。要想把变量 x, y 和 z 全部声明为整数引用时,应该使用 `int &x = a, &y = c, &z = c` 这 3 个表达式,而不应该使用不正确的表达式 `int &x = a, y = b, z = c`,或者其他常见的错误表达式 `int &x, y, z`。

函数可以返回引用,但比较危险。在被调用函数中声明的变量返回引用时,在被调用函数内,应将变量声明为 `static`。否则,引用会引用函数中止时已被删除的自动化变量;此类变量被称为“未定义变量”,程序将出现意外行为(此时,有的编译器会发出警告消息)。对未定义变量的引用被称为“悬挂引用”。

**常见编程错误 3.28** 声明引用变量时未将其初始化是语法错误。

**常见编程错误 3.29** 试图将一个已声明的引用重新指定为另一个变量的别名是个逻辑错误。另一个变量的值被分配给已作为别名的引用所在的地址。

**常见编程错误 3.30** 向被调用函数中的自动变量返回指针或引用是逻辑错误。对此,有的编译器会发出警告消息。

### 3.18 默认实参

函数调用通常传递参数的特定值。程序员可以指定此类参数是“默认实参”(default argument),程序员也可以为此类参数提供默认值。函数调用中省略默认实参时,编译器或自动插入该参数的默认值,并将其传递到函数调用中。

默认实参必须是函数参数列表中最右边(追尾的)的那个参数。在带有两个或更多默认实参的调用中,如果省略的实参不是参数列表中最右边的参数,那么该参数右侧的所有参数也必须省略。默认实参应该在函数名首次出现时指定——通常是在函数原型中。默认实参可以是常量、全局变量或函数调用。默认实参还可以随内联函数一起使用。

图 3.23 演示了默认实参在计算箱子容积时使用。第 8 行中,boxVolume 的函数原型指定 3 个实参的默认值都是 1。注意,默认值只能在函数原型中定义。此外还须注意,为增强程序的可读性,我们在函数原型中提供了变量名称。一般说来,变量名称在函数原型中并不是必须的。

```

1  //Fig. 3.23; fig03_23.cpp
2  //Using default arguments
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12     cout << "The default box volume is: " << boxVolume()
13         << "\n\nThe volume of a box with length 10,\n"
14         << "width 1 and height 1 is: " << boxVolume( 10 )
15         << "\n\nThe volume of a box with length 10,\n"
16         << "width 5 and height 1 is: " << boxVolume( 10, 5 )
17         << "\n\nThe volume of a box with length 10,\n"
18         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
19         << endl;
20
21     return 0;
22 }
23
24 //Calculate the volume of a box
25 int boxVolume( int length, int width, int height )
26 {
27     return length * width * height;
28 }
```

输出结果:

The default box volume is:1

The volume of a box with length 10,

```
width 1 and height 1 is:10
The volume of a box with length 10,
width 5 and height 1 is:50
The volume of a box with length 10,
width 5 and height 2 is:100
```

图 3.23 使用默认实参

对 `boxVolume` 第一次调用(第 12 行)没有指定实参,因此全部采用了 3 个默认值。第二次调用(第 14 行)传递了一个 `length` 实参,因此使用了 `width` 和 `height` 实参的默认值。第 3 次调用(第 16 行)传递了 `length` 和 `width` 的实参,所以使用了 `height` 实参的默认值。最后一次调用(第 18 行)传递了 `length`, `width` 和 `height` 的实参,所以名为使用默认值。

**良好编程习惯 3.12** 使用默认实参可以简化函数调用的编程工作。但也有的程序员认为,显式指定所有实参会使程序更清晰。

**常见编程错误 3.31** 指定并打算使用非最右边(追尾的)实参的默认实参(也就没有把所有最右边的实参指定为默认实参)是语法错误。

## 3.19 一元作用域分辨符

声明同名局部变量和全局变量是可能的。C++ 提供了一元作用域分辨符(`::`)(unary scope resolution operator),以便在具有同名局部变量的作用域中访问全局变量。一元作用域分辨符不能用于访问外层块中的同名局部变量。如果全局变量名与作用域中的局部变量名不同,就可以在不用一元作用域分辨符的情况下,直接访问全局变量。第 6 章,我们将讨论随同类一起使用的“二元作用域分辨符”。

图 3.24 演示了其局部和全局变量名称相同的一元作用域分辨符。为了强调常量变量 `PI` 的局部和全局版本之间的区别,程序将其中一个声明为 `double`,另一个声明为 `float`。

```
1 //Fig.3.24: fig03_24.cpp
2 //Using the unary scope resolution operator
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setprecision;
11
12 const double PI = 3.14159265358979;
13
14 int main()
15 {
16     const float PI = static_cast<float>( ::PI );
17
18     cout << setprecision( 20 )
```

```

19         << " Local float value of PI = " << PI
20         << " \nGlobal double value of PI = " << ::PI << endl;
21
22     return 0;
23 }

```

Borland C++ 命令行编译器输出的错误消息:

```

Local float value of PI = 3.141592741012573242
Global double value of PI = 3.141592653589790007

```

Microsoft Visual C++ 编译器输出的错误消息:

```

Local float value of PI = 3.1415927410125732
Global double value of PI = 3.14159265358979

```

图 3.24 使用一元作用域分辨符

**常见编程错误 3.32** 试图利用一元作用域分辨符来访问外层块中的非全局变量时,如果外层块中没有与该变量同名的全局变量,就会出现语法错误,反之,如果有与该变量同名的全局变量,则会出现逻辑错误。

**良好编程习惯 3.13** 尽量避免在程序中出现用途不同的同名变量。尽管在许多情形下允许这样做,但这样往往会产生混乱。

## 3.20 函数重载

C++ 允许将不同的函数定义为同样的名称,只要这些函数拥有不同的参数集(至少这些参数的类型是不同的),这个功能被称为“函数重载”(function overloading)。调用重载函数时,C++ 编译器会检查调用中的参数个数、类型和顺序来选择恰当的函数。函数重载通常用于创建用于执行类似任务的同名、但具有不同数据类型的函数。

**良好编程习惯 3.14** 执行类似任务的函数重载功能可以令程序更易于阅读和理解。

图 3.25 采用重载函数 square 来计算 int 类型值的平方和 double 类型值的平方。在第 8 章,我们将讨论如何重载操作符,从而定义其如何操作用户自定义数据类型的对象(事实上,在此之前,我们使用了许多重载操作符,其中包括流插入操作符 << 和流提取操作符 >>。我们还将第 8 章介绍重载 << 和 >>)。3.21 小节要介绍函数模板,这些模板用于自动生成对不同数据类型数值执行相同任务的重载函数。第 12 章将详细讨论函数模板和类模板。

```

1 //Fig.3.25: fig03_25.cpp
2 //Using overloaded functions
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int square( int x ) { return x * x; }
9
10 double square( double y ) { return y * y; }
11

```

```

12 int main()
13 {
14     cout << "The square of integer 7 is" << square( 7 )
15         << " \nThe square of double 7.5 is " << square( 7.5 )
16         << endl;
17
18     return 0;
19 }

```

输出结果:

The square of integer 7 is 49

The square of double 7.5 is 56.25

图 3.25 重载函数用法示例

重载函数之间的区别在于其签名 (signatures)——签名是函数名称及其参数类型的组合。为保证类型安全链接 (type-safe linkage), 编译器利用每个函数的参数个数和类型对其标识符实行编码 (有时也被称为名称改编 (name mangling) 和名称修饰 (name decoration))。类型安全链接可以保证调用恰当的重载函数, 以及形参和实参之间的对应。编译器能够侦测和报告链接错误。图 3.26 中的程序是在 Borland C++ 编译器上编译的, 图中没有像以往那样展示输出结果, 我们展示了改编后的函数名称, 这一名称是 Borland C++ 编译器用汇编语言来生成的。每个改编后的名称前面都有一个 @, 然后紧跟函数名。改变后的参数列表以 \$ 开头。在函数 nothing2 的参数列表中, c 代表 char 类型的值, i 代表 int 类型的值, pf 代表 float \* 类型的值, pd 代表 double \* 类型的值。在函数 nothing1 的参数列表中, i 代表 int 类型的值, f 代表 float 类型的值, c 代表 char 类型的值, pi 代表 int \* 类型的值。这两个 square 函数是通过其参数列表来区分的; 一个指定 d 来代表 double 类型的值, 另一个指定 i 来代表 int 类型的值。改编后的函数名称中不会指定返回类型。函数名称改编是只与编译器有关。重载函数可以有不同类型的返回值, 但必须有不同的参数表。

```

1 //Fig. 3.26; fig03_26.cpp
2 //Name mangling
3
4 int square( int x ) { return x * x; }
5
6 double square( double y ) { return y * y; }
7
8 void nothing1( int a, float b, char c, int *d )
9     { } //empty function body
10
11 char *nothing2( char a, int b, float *c, double *d )
12     { return 0; }
13
14 int main()
15 {
16     return 0;
17 }

```

输出结果:



```

_main
@ nothing2 $ qcipfpd
@ nothing1 $ qifcpi
square $ qd
@ square $ qi

```

图 3.26 改编函数名称以保证类型安全链接

**常见编程错误 3.33** 创建参数列表相同,而返回类型不同的重载函数是语法错误。

编译器只通过参数列表来区分同名函数。重载函数不要求参数个数一定相同。在重载带有默认参数的函数时,程序员应该提高警惕,以免产生歧义。

**常见编程错误 3.34** 像调用另一个重载函数那样调用省略默认参数的函数会产生语法错误。例如在程序中,如果同时包含显式不取参数的函数和另一个其中带有所有默认参数的同名函数,那么打算在没有传递任何参数的调用中使用这个函数名就会产生语法错误。

## 3.21 函数模板

重载函数通常用于执行类似的运算,这些运算涉及到应用于不同数据类型的不同程序逻辑。如果程序逻辑和运算对于各个数据类型都是一样的,那么利用函数模板(function templates)来执行就会非常简洁而方便。程序员只需写一个函数模板定义即可。以函数调用中提供的参数类型为例,C++ 会自动生成单独的模板函数(template function)对各种类型的调用进行相应的处理。因此,定义一个函数模板就相当于定义了整套解决方案。

所有函数模板定义都是以关键字 `template` 开头,随后紧跟该函数模板的形式参数列表,该列表用尖括号(< 和 >)括起来。每个形式参数之前不是关键字 `typename` 就是关键字 `class`。形式参数(formly type parameter)是内置类型或用户自定义类型,用于指定函数参数的类型,指定函数的返回类型,以及用于在函数定义体中声明变量。随后是函数定义,而且定义方式和其他函数是一样的。

图 3.27 中的程序使用了如下所示的函数模板定义

```

template < class T >          //or template< typename T >
T maximum( T value1, T value2, T value3 )
{
    T max = value1;
    if ( value2 > max )
        max = value2;
    if ( value3 > max )
        max = value3;
    return max;
}

```

这个函数模板把一个形式参数 `T` 声明为函数 `maximum` 要测试的数据类型。当编译器在程序源代码中检测到 `maximum` 调用时,传递给 `maximum` 的数据类型就会通过模板定义提交给参数 `T`,而C++ 会创建一个完整的函数,令其检查这 3 个指定数据类型值的最大值。然后,对创新

建的函数进行编译。因此,我们说模板实际上是某种形式的代码生成工具。在图 3.27 的程序中,创建了 3 个函数——一个要取 3 个 int 值,一个要取 3 个 double 值,还有一个要取 3 个 char 值。下面便是为 int 类型创建的函数模板

```
int maximum( int value1, int value2, int value3)
{
    int max = value1;
    if (value2 > max)
        max = value2;
    if (value3 > max)
        max = value3;
    return max;
}
```

特定的模板定义中,正式参数列表内的类型参数名称必须是独一无二的。图 3.27 解释了如何用 maximum 模板函数来确定 3 个 int 值、3 个 double 值和 3 个 char 值的最大值。

```
1 //Fig.3.27: fig03_27.cpp
2 //Using a function template
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 template < class T >
10 T maximum( T value1, T value2, T value3 )
11 {
12     T max = value1;
13
14     if ( value2 > max )
15         max = value2;
16
17     if ( value3 > max )
18         max = value3;
19
20     return max;
21 }
22
23 int main()
24 {
25     int int1, int2, int3;
26
27     cout << "Input three integer values: ";
28     cin >> int1 >> int2 >> int3;
29     cout << "The maximum integer value is: "
30         << maximum( int1, int2, int3 );    //int version
31
32     double double1, double2, double3;
33
```

```

34     cout << "\nInput three double values: ";
35     cin >> double1 >> double2 >> double3;
36     cout << "The maximum double value is: "
37         << maximum( double1, double2, double3 ); //double version
38
39     char char1, char2, char3;
40
41     cout << "\nInput three characters: ";
42     cin >> char1 >> char2 >> char3;
43     cout << "The maximum character value is: "
44         << maximum( char1, char2, char3 )      //char version
45         << endl;
46
47     return 0;
48 }

```

输出结果:

```

Input three integer values:1 2 3
The maximum integer value is:3
Input three double values:3.3 2.2 1.1
The maximum double value is:3.3
Input three characters:A C B
The maximum character value is:C

```

图 3.27 使用函数模板

**常见编程错误 3.35** 在函数模板的每个类型参数前,不加上关键字 `class` 或 `typename` 是语法错误。

## 3.22 【可选案例分析】对象思想:标识类的属性

在第2章的“对象思想”一节,我们针对电梯模拟程序,进入了面向对象设计的第一阶段,即确定实现电梯模拟程序所需要的类。一开始,我们列出了问题陈述中的所有名词,而且为每类名词创建了单独的类,令其在电梯模拟程序中分担一部分责任。然后,我们在UML类图中展示了这些类及其关系,类具有各自的属性和操作。在C++程序中,类属性以数据的形式实现;类的操作则以函数的形式实现。在本节,我们要确定实现电梯模拟程序所需的类的属性。第4章,我们则要确定类的操作是什么。第5章,我们会集中精力讨论电梯模拟程序中不同对象之间的交互,也就是人们常说的合作。

以某些实际对象的属性为例。人的属性包括身高和体重。收音机的属性包括电台设置、音量设置和是采用调频(FM)还是调幅(AM)。汽车的属性包括速度表和里程表、剩余油量、档位等。个人电脑的属性包括制造商(比如苹果公司、IBM还是康柏)、显示器类型(比如单显还是彩显)、主存储器的大小(以MB为单位)和硬盘容量大小(以GB为单位)等等。

属性是对类的描述。我们可以在问题陈述中查找具有描述性的词句来获得系统的属性。针对我们找到的每个具有描述性的词句,我们为其创建了一个属性,并将其分配给一个类。我们还创建了可以代表类所需数据的属性。例如,Scheduler类需要知道在什么时间创建下一个人,并令其走到每一个楼层。在图3.28的表格中,列出了问题陈述中的单词或短

语,它们对每个类进行了描述。

类	描述性单词和短语
Elevator	电梯每天在第1层关门等待 改变方向:向上和向下移动 载客量为一人 花5秒钟从一层移向另一层
Clock	每一天的时间开始设为0
Scheduler	[安排一个人的抵达时间]针对每一层,创建下一个随机抵达时间(从现在算起的5~20秒的时间内)
Person	人的编号(来自输出)
Floor	载客量为一人 已被占据/未被占据
FloorButton	已被按下
ElevatorButton	已被按下
Door	门被打开/关闭
Bell	问题陈述中无描述性词句
Light	灯关/开
Building	问题陈述中无描述性词句

图 3.28 问题陈述中描述性的单词和短语

注意 Bell 和 Building 类没有属性。在继续案例分析的过程中,我们会不断地添加、修改和删除与电梯系统中各个类相关的信息。

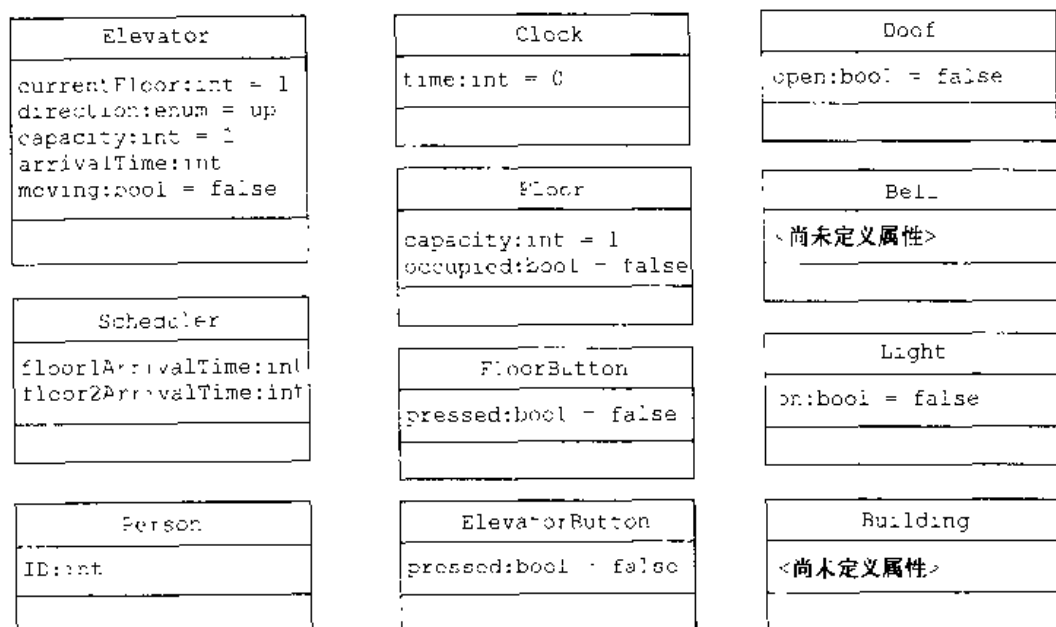


图 3.29 显示了属性的类图

图 3.29 是一个类图,其中列出了电梯系统中各个类的部分属性——这些属性是根据图 3.28 中的词汇来创建的。在 UML 类图示,类的属性放在类矩形的中部一格中。以 Elevator 类的以下属性

```
capacity : int = 1
```

为例,其中包含了同属性有关的 3 个方面的信息。属性有一个名称,这里为 `capacity`。属性要有一个类型,这里为 `int`。类型要依赖于用于编写软件系统的语言。比如在 C++ 中,这个值可以是一个基本类型,比如 `int`, `char` 或 `float`;也可采用像一个类那样的用户自定义类型(我们将在第 6 章开始对类进行研究。届时会了解到,每个新类在本质上都是一种新数据类型)。

还可为每个属性指定一个初始值。`capacity` 属性的初始值是 1。假如一个特定的属性没有特殊的初始值,那么只有它的名字和类型(用一个冒号分隔)才会显示。例如, `Scheduler` 类的 `floorArrivalTime` 属性具有 `int` 类型。在此,我们不指定初始值,由于这个属性的值是一个我们现在还不知道的随机数;随机数在执行期间才能决定。就目前来说,不必过份关心属性的类型或初始值的问题。我们只包括能从问题陈述中收集到的信息。

### 3.22.1 状态图

系统中的对象都有状态。状态描述了在一个给定时刻,一个对象的状况是什么。状态图为我们提供了一种简便的方式,可表示对象在一个系统中,如何(以及在什么条件下)更改其状态。

图 3.30 是一个简单的状态图,它建模了 `FloorButton` 类或 `ElevatorButton` 类的一个对象的状态。状态图中的每种状态都可表示成一个圆角矩形,并在内部写上状态名。从一个实心圆会发出一条箭头,指向其初始状态(即“Not pressed”状态)。带箭头的实线表明两种状态间的转变。一个对象可从一种状态转变成另一种状态,以响应一个事件。例如, `FloorButton` 和 `ElevatorButton` 这两个按钮类会从“Not pressed”(未按下)状态转变成“Pressed”(已按下)状态,以响应一个“按钮被按下”事件。导致这种转变的事件名要写在同那个转变对应的箭头线旁边(可在此包括更多与事件有关的信息,详情参见后文描述)。

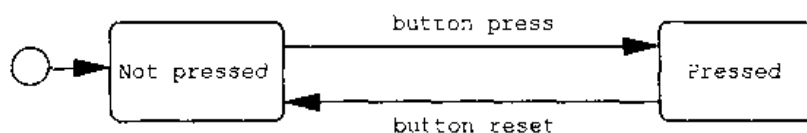


图 3.30 `FloorButton` 和 `ElevatorButton` 类的状态图

图 3.31 展示了 `Elevator` 类的状态图。电梯有 3 种可能的状态:“Waiting”、“Servicing Floor”(即电梯停在某一层,但正在忙于重置电梯按钮,或者同楼层通信等等)和“Moving”。电梯最开始处在“Waiting”状态。至于造成状态转变的事件,则在恰当的转变线旁边进行了标注。

接下来讨论这个状态图中的事件。以下文字

```
button press {need to move}
```

告诉我们“按钮被按下”事件会导致电梯从“Servicing Floor”(为楼层提供服务)状态转变成“Moving”(移动)状态。方括号中的防卫条件指出,只有在电梯“需要移动”的前提下,才会发生状态的转变。整个事件文本指出,只有在电梯需要移动的时候,电梯才会从“Servicing Floor”状态转变成“Moving”状态,以响应“button pressed”事件。类似地,一旦在电梯当前所在的楼层按下一个按钮,电梯会从“Waiting”状态转变成“Servicing Floor”状态。

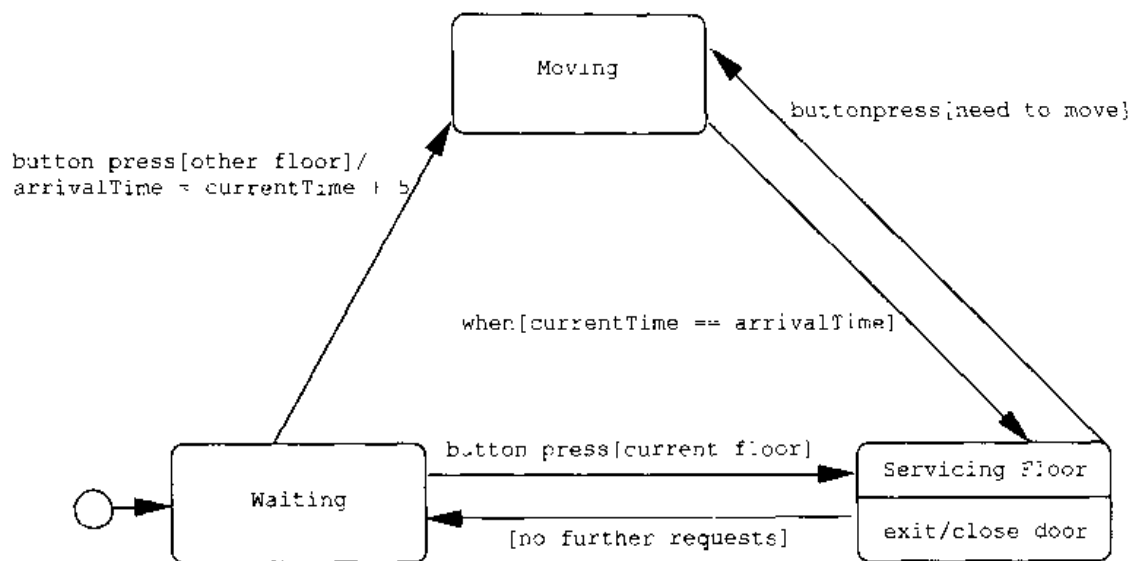


图 3.31 Elevator 类的状态图

在从“Waiting”状态转变成“Moving”状态的转变线旁边的文本中,指出只有在按钮被按下时(假如在另一层按下了按钮),才会发生这样的状态转变。正斜杠(“/”)指出随同这种状态转变,会采取一项行动。在此,电梯会采取“计算并设置抵达另一层的时间”的行动。<sup>①</sup>

特定的条件为 true 时,也可能生一次状态转变。以下文字

when[currentTime == arrivalTime]

指出,一旦当前模拟时间等于电梯安排抵达某一层的时间,电梯就会从“Moving”状态转变成“Servicing Floor”状态。

在从“Servicing Floor”状态转变成“Waiting”状态的转变线的旁边,标注的文本指出,在没有其他请求电梯提供服务的情况下,电梯就会从“Servicing Floor”状态转变成“Waiting”状态。<sup>②</sup>

一个对象处在特定的状态时,也可以采取特定的行动(参见图 3.31 中的“Servicing Floor”状态)。为了对这些行动进行建模,我们需要将恰当的状态分割成两格。在顶部一格中,包含了状态名;在底部一格中,则包含了状态行动。UML 定义了一个特殊的行动标签,名为 exit。exit 行动表明,当对象脱离(exit)一种状态时,应采取一项行动。在我们的模型中,在电梯脱离“Servicing Floor”状态时,会采取“close door”(关门)行动。换言之,一旦电梯需要移动,那么首先必须关门;假如电梯不需要向其他请求(按钮被按下)提供服务,那么关门,并进入“Waiting”状态。

① 在现实世界的电梯系统中,电梯上的一个传感器可能会造成它停在某一层。在我们的电梯模拟程序中,已知从一层移到另一层电梯会花 5 秒钟。所以,电梯只需设定抵达一个楼层的时间即可。而且在事先设定的那个时间,电梯肯定会停下来。

② 在现实世界的电梯系统中,电梯可能在经过了一段特定的超时时间后,自动地在这两种状态之间切换。我们想为一个模拟程序编程,但不想过分地关心电梯如何才能“知道”何时没有针对其服务的更多请求。所以,我们只是简单地说,电梯会在没有更多请求存在的情况下,改变其状态。

### 3.22.2 活动图

活动图其实是状态图的一种变化形式。活动图将重点集中在一个对象所采取的行动上面;换言之,活动图建模的是对象在其生命期内的全部活动。

根据图 3.31 的状态图,我们并不知道一旦建筑物内两个不同的人在不同的楼层同时按下楼层按钮,那么电梯的状态会变成什么。另外,从中也看不出电梯如何决定它是否需要移动。图 3.32 的活动图则对状态图的信息进行了补充,对电梯用于响应一个服务请求的活动进行了建模。

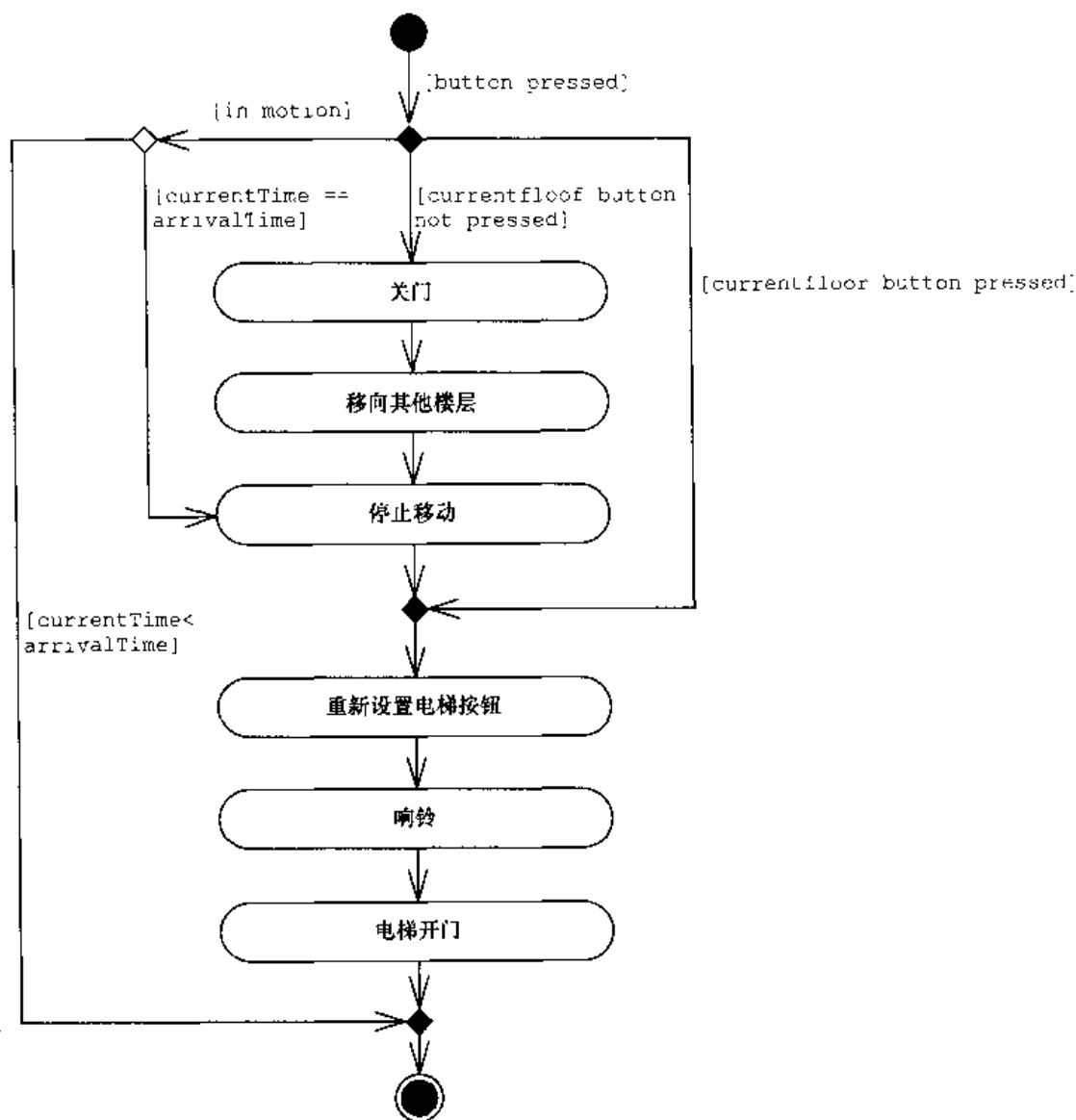


图 3.32 对用于响应“button pressed”事件的电梯逻辑用活动图建模

活动是用椭圆表示的,活动的名称置于椭圆内部;实箭头线用于连接两个活动,并指出活动发生的先后顺序。和状态图一样,实心圆表明一系列活动发生的起点。在这个图中,一系列活动是在一个按钮被按下时开始执行的(换言之,假如任何一层的楼层按钮进入

“Pressed”状态)。一旦条件为 true, 电梯必须做出决定, 它是用一个菱形表示的<sup>①</sup>。此时, 电梯会在不同的活动中挑选一个, 具体取决于特定的条件是什么。从菱形延展出来的每一条线(或者路径)都表示一系列不同的活动。在每个路径的旁边, 会放置一个防卫条件, 表明在什么情况下才会执行此路径。

在我们的图示中, 电梯一旦发现有人按按钮, 就会执行 3 个不同的活动之一。假如电梯正在移动(即处在“Moving”状态), 那么电梯不能马上执行更多的行动, 所以当前路径上的活动序列只是简单地中止。被一个圆套着的实心圆(有时叫做“牛眼”)指明一个活动图的结束位置。

假如在电梯所在的当前楼层按下楼层按钮, 电梯会重置它的按钮, 响铃, 并开门。假如电梯所在的当前楼层的按钮未被按下, 那么电梯首先必须关门, 移到另一个楼层, 在那里停下, 然后才能为另一楼层提供服务。注意 UML 要求使用另一个小菱形符号对决定路径进行合并。一旦电梯开门, 活动序列就可以中止了。

### 3.22.3 本节要点

本节我们对系统中有关类的知识进行了扩充(在后面几章里, 还会继续扩充), 并在我们的类图中, 对新学到的知识进行了表示。我们还使用了状态图和活动图来获得与系统工作方式有关的更多信息。即使至今仍未开始用 C++ 接触面向对象编程的细节, 但我们其实已掌握了数量可观的信息, 它们对我们的系统进行了充分的描述。在第 4 章和第 5 章末尾的“对象思想”小节, 还会讨论同类有关的操作, 以及类与类之间如何交互(即如何“合作”)。

**说明:**本章介绍了如何实现“随机性”。语句

```
arrivalTime = currentTime + ( 5 + rand() % 16 );
```

可用于随机安排下一个人进入楼层的时间。

## 3.23 小结

- 开发和维护大型程序的最佳方式是将其分成更小的程序模块, 与原始程序相比, 每个小模块将更易于管理。在 C++ 中, 模块被称为类和函数。
- 函数的调用由函数调用来实现。函数调用通过名称来指定函数, 并提供被调用函数执行其任务时所需的信息(比如参数)。
- 信息隐藏的的目的是让函数只能访问完成任务时所需的信息。这是实现最低权限的一种方式, 也是实现良好软件工程所需的最重要的原则之一。
- double 和 float 一样, 都是浮点类型。但和 float 类型的变量相比, double 类型的变量可以保存更大、精度更高的数值。
- 函数的各个参数可以是常量、变量或者是表达式。
- 局部变量只适用于函数定义。函数无法得知其他函数的实现细节(包括局部变量在内)。

<sup>①</sup> 这个符号不应同流程图中使用的大菱形符号混淆起来(如 2.21 节展示的那样)。



- 函数定义的常见格式为

```
return-value-type function-name( parameter-list )
{
    Declaration and statements
}
```

- return-value-type(返回值类型)指的是返回调用函数的值的类型。如果函数没有返回值,那么 return-value-type 就会被声明为 void。function-name(函数名)是任意一个有效的标识符。parameter-list(参数列表)是一个用逗号隔开的列表,其中包含将被传递给函数的变量的声明。如果函数不接收任何值,parameter-list 就会被声明为 void。function-body(函数体)是组成函数的声明和语句的集合。
- 传递给函数的参数与函数定义中的参数在个数、类型和顺序上应该一致。
- 程序碰到函数调用时,控制权会从调用点转移到被调用的函数,执行了该函数之后,控制权再返回调用者。
- 被调用函数可以 3 种方式把控制权返回调用者。如果函数不没有返回值,控制权就会在到达函数结束时的右花括号或在执行语句

```
return;
```

时返回。如果函数返回了结果,语句

```
return expression;
```

就会返回 expression 的结果值。

- 函数原型声明了函数的返回类型、函数希望接收的参数个数、类型和顺序。
- 函数原型让编译器能够验证函数是否被正确调用。
- 编译器会忽略函数原型中提及的变量名。
- 每个标准库都有一个相应的头文件,库中所有函数的函数原型和这些函数所需的符号常量定义都包含在这个头文件中。
- 程序员可以也应该创建并包含他们自己的头文件。
- 通过传值调用传递参数时,会产生变量值的一个副本,副本会被传递给被调用的函数。对被调用函数中副本的更改不会对变量的原始值产生影响。
- rand 函数生成了 0 ~ RAND\_MAX 之间的一个整数,这个值至少应为 32 767。
- rand 和 srand 的函数原型包含在 <cstdlib> 内。
- 通过比例缩放和位移,rand 生成的值可以在特定的范围内,生成新值。
- 要想随机化 rand 的结果输出,需用标准库函数 srand。
- srand 语句一般只能在程序经过彻底调试后,才能插入程序中。调试期间,最好忽略 srand。这样可保证重用性,这实际上相当于保证修改后的随机数生成程序能正常运行。
- 要想在无需每次输入种子数的情况下实现随机化,应使用 srand( time(0) )。函数 time 通常会返回以秒计的“日历数”。函数 time 的原型包含在头文件 <ctime> 内。
- 按比例缩放和位移随机数的常见等式为

```
n = a + rand() % b;
```

a 代表的是位移值(等于所需要的连续整数范围内的第一个数值),b 指的是缩放因

子(等于所需要的连续整数范围的宽度)。

- 以关键字 `enum` 开头,随后紧跟类型名称的枚举是一个用标识符表示的整数常量的集合。
- 除非特别说明,不然这些枚举常量的值会从 0 开始,而且增量为 1。
- `enum` 中的标识符必须是惟一的,但不同枚举常量可以取同样的整数值。
- 在枚举定义中,任何一个枚举常量都可以显式分配到一个整数值。
- 每个变量标识符都有保存类、作用域和连接这几个属性。
- C++ 提供了 5 个存储类说明符: `auto`, `register`, `extern`, `mutable` 和 `static`。
- 标识符的存储类决定了标识符在内存中存在的时间。
- 标识符的作用域是程序中可以引用该标识符的地方。
- 标识符的连接属性决定了多源文件程序中,只在当前源文件,还是在具有正确声明的任何一个源文件中识别这个标识符。
- 自动存储类变量是在进入声明该变量的程序块时创建,它们存在于程序块活动期间,并在退出程序块时删除。在默认状态下,函数的局部变量通常都是自动存储类。
- 存储类说明符 `register` 可以放在自动变量声明之前,让编译器在计算机的高速寄存器中维护该变量。编译器可能会忽略 `register` 声明。关键字 `register` 只能用于自动存储类变量。
- 关键字 `extern` 和 `static` 是用于声明静态存储类变量和函数的说明符。
- 程序开始执行时,就会分配和初始化静态存储类变量。
- 两类标识符有静态存储类:外部标识符和存储类说明符 `static` 中声明的局部变量标识符。
- 全局变量的创建是这样的:把变量声明放在任何一个函数定义以外,而且整个程序的执行期间,全局变量中的值会保持不变。
- 声明为 `static` 的局部变量在退出声明它们的函数时,仍然会维持原值不变。
- 对于静态存储类的所有数字变量,如果程序员不显式对其初始化,它们会被初始化为 0。
- 标识符作用域包括函数范围、文件范围、块范围和函数原型范围。
- 标号是惟一具有函数范围的标识符。标号可用于其所在函数中的任何地方,但不能在函数体以外引用。
- 在任何函数外部声明的标识符都有文件范围。比如可以从声明标识符到文件末尾处的任何函数中访问标识符。
- 在程序块内部声明的标识符有块范围。块范围在表示程序块结束的右花括号处结束。
- 函数头部声明的局部变量有块范围,它和函数参数一样,也被函数视为局部变量。
- 任何程序块都包含变量声明。嵌套程序块时,外部块中的标识符与内部块中的标识符会同名,在内部块中止之前,外部块中的标识符会处于“隐蔽状态”。
- 只有函数原型参数列表中的标识符才是惟一具有函数原型作用域的标识符。函数原型中的标识符可重复用于程序中的其他地方,而不会产生歧义。

- 递归函数是直接或间接调用其自身的函数。
- 如果递归函数是通过基本情况调用的,那么函数只会返回一个结果。如果函数是通过更复杂的问题来调用的,那么函数就会把问题分解为两个概念性的部分:函数知道如何处理的部分和小于原问题的更小部分(即函数无法处理的部分)。由于新生成的问题与原问题类似,所以,函数会发起递归调用来处理小问题。
- 要想中止递归调用,那么每次递归函数调用其自身来处理原问题的更小部分时,所产生的问题都要越来越小,直到可以通过基本情况来解决问题为止。函数识别出基本情况时,就会把结果返回前一次函数调用,并将后续的结果一一返回,直到最初的函数调用最后终于返回最终结果。
- C++ 标准库没有指定大部分操作数的求值顺序。C++ 指定了操作符 `&&`、`||`、逗号 `(,)` 操作符和 `?:` 的求值顺序。前 3 个是二进制操作符,其操作数的求值顺序是从左到右。最后一个操作符是 C++ 中独有的三元操作符。对于这种操作符,应先求值最左边的操作数;如果最左边的操作数非零,就接着求值中间的操作数,而忽略最后的操作数;如果最左边的操作数为零,接着求值的将是第 3 个操作数,而忽略中间的操作数。
- 迭代和递归都是基于控制结构的:迭代使用的是重复结构;递归使用的则是选择结构。
- 迭代和递归两者都涉及到了重复:迭代显式使用重复结构;递归则通过重复性的函数调用来实现重复。
- 迭代和递归均涉及到中止测试:迭代在循环条件失败时中止;递归则在识别出基本情况时中止。
- 迭代和递归会发生无休止的情况;如果循环测试永远不变成 `false`,迭代就会产生无限循环;如果递归步骤永远不能回推到基本情况,就会陷入无穷递归。
- 重复性递归调用机制会不断增加函数调用开销。这会大量占用处理器时间和内存空间。
- C++ 程序不会编译函数,除非为每个函数提供函数原型或在首次调用函数之前,先定义该函数。
- 对于没有返回值的函数,要将其返回类型声明为 `void`。在函数调用中,试图通过函数返回值或函数调用产生的结果值是语法错误。空参数列表用空的圆括号 `()` 或圆括号内的 `void` 来指定。
- 内联函数消除了函数调用的开销。程序员用关键字 `inline`,建议编译器在可能的情况下,在一行内生成函数代码(也就是复制函数代码),借此减少函数调用。编译器可以选择忽略内联建议。
- C++ 提供了利用引用参数实现直接按引用调用的方式。为了指明函数参数是按引用进行传递的,需要在函数原型中的参数类型后加一个 `&`。在函数调用中,通过名称来提及变量,那么该变量就会按引用调用来传递。在被调用函数中,通过其名称来提及变量实际上指的是调用函数中的原始变量。因此,被调用函数可以直接修改原始变量。

- 引用参数还可以用作函数内部其他变量的别名。引用参数必须在其声明中得以初始化,而且它们还不能作为其他变量的别名重新赋值。一旦引用变量被声明为另一个变量的别名,对该别名的所有操作实际上是对原变量本身进行的。
- C++ 允许程序员指定默认参数及其默认值。如果函数调用中忽略了默认参数,就会是默认参数的默认值。默认参数必须是函数参数列表中最右边(追尾的)的那个参数。默认参数应该在函数名首次出现时指定。默认值可以是常量、全局变量和函数调用。
- 二元作用域分辨符(::)令程序能访问与局部变量同名的域内全局变量。
- 可以用同样的名称,不同类型的参数来定义多个函数。这种方法叫作函数重载。调用重载函数时,编译器会根据函数调用中的参数个数和类型来选择恰当的函数。
- 重载函数可以有不同的返回值,但必须有不同的参数列表。仅根据返回类型来区分函数会造成编译错误。
- 利用函数模板,可以创建针对不同类型的数据执行相同操作的函数,但函数模板只需定义一次。

## 本章术语

ampersand (&) suffix & 前缀

argument in a function call 函数调用中的参数

auto storage class specifier 自动存储类说明符

automatic storage 自动存储

automatic storage class 自动存储类

automatic variable 自动变量

base case in recursion 递归中的基本情况

block 块

block scope 块范围

C++ standard library C++ 标准库

call a function 调用函数

call-by-reference 按引用调用

call-by-value 按值调用

called function 被调用函数

caller 调用者

calling function 调用函数

coercion of arguments 强制参数类型转换

collaboration 合作

component 组件

const variable 常量变量

copy of a value 值的副本

dangling reference 悬挂引用

default function arguments 默认函数参数

divide and conquer 分而治之

element of chance 机会元素

enumeration 枚举

enumeration constant 枚举变量

extern storage class specifier extern 存储类说明符

factorial function 阶乘函数

file scope 文件作用域

function 函数

function call 函数调用

function declaration 函数声明

function definition 函数定义

function overloading 函数重载

function prototype 函数原型

function scope 函数范围

function signature 函数签名

global variable 全局变量

header file 头文件

infinite recursion 无穷递归

information hiding 信息隐藏

inline function 内联函数

invoke a function 调用函数

iteration 迭代

linkage 链接

linkage specification 链接规范

local variable 局部变量

math library functions 数学库函数

mixed-type expression 混合类型表达式

modular program 混合类型表达式

mutable storage class specifier mutable 存储类说明符

name decoration 名字修饰  
 name mangling 名字改编  
 named constant 命名常量  
 optimizing compiler 优化编译器  
 overloading 重载  
 parameter in a function definition 函数定义中的参数  
 principle of least privilege 最低权限原则  
 programmer-defined function 程序员定义的函数  
 promotion hierarchy 层次提升  
 random number generation 随机数生成  
 randomize 随机化  
 read-only variable 只读变量  
 recursion 递归  
 recursive call 递归调用  
 recursive function 递归函数  
 reference parameter 引用参数  
 reference type 引用类型  
 register storage class specifier register 存储类说明符  
 return-value type 返回值类型  
 scaling 比例缩放  
 scope 作用域(范围)  
 shifting 位移(移位)  
 side effect 副作用  
 signature 签名  
 simulation 模拟  
 software engineering 软件工程  
 software reusability 软件重用  
 standard library header files 标准库头文件  
 static storage class specifier static 存储类说明符  
 static storage duration 静态存储期  
 static variable static 变量  
 storage class specifier 存储类说明符  
 storage class 存储类  
 template 模板  
 template function 模板函数  
 type-safe linkage 类型安全链接  
 unary scope resolution operator(::)  
 一元作用域分辨符(::)

## “对象思想”术语

action 行动  
 action-label 行动级  
 activity 活动  
 activity diagram 活动图  
 arrowhead symbol in the UML UML 中的箭头符号  
 attribute 属性  
 attribute initial value in the UML  
 UML 中的属性初始值  
 attribute name in the UML UML 中的属性名  
 attribute type in the UML UML 中的属性类型  
 “bull’s-eye” symbol in UML activity diagram  
 UML 活动图中的“牛眼符号”  
 descriptive words in problem statement  
 问题陈述中的描述性词句  
 diamond symbol in UML activity diagram  
 UML 活动图中的菱形符号  
 event 事件  
 “exit” action “exit”行动  
 guard condition 防卫条件  
 initial state 初始状态  
 initial value of class attribute 类属性的初始值  
 oval symbol in UML activity diagram  
 UML 活动图中的椭圆符号  
 rounded rectangle symbol in UML statechart diagram  
 UML 状态图中的圆角矩形符号  
 solid line with arrowhead symbol in UML statechart and  
 activity diagrams  
 UML 状态和活动图中的箭头实线  
 starting point symbol in UML statechart and activity dia-  
 grams UML 状态图和活动图中的起始符  
 state 状态  
 statechart diagram 状态图  
 transition 转变  
 “when” event “when”事件

## 常见编程错误

3.1 使用数学库函数时,忘记将数学头文件包含在内是语法错误。程序中所用的每一个标准库函数都必须包含标准的头文件。

- 3.2 函数定义中省略返回值类型是语法错误。
- 3.3 忘记在需要返回值的函数中返回返回值是语法错误。
- 3.4 在返回值类型已被声明为 void 的函数中返回返回值是语法错误。
- 3.5 将同一类型的函数参数声明为 float x,y,而不是 float x,float y。参数声明 float x,y 实际上会报告编译出错。因为参数列表中每个参数都需要指明类型。
- 3.6 在函数定义的参数列表右括号之后添加分号是语法错误。
- 3.7 将函数参数再次定义为函数中的局部变量是语法错误。
- 3.8 函数调用中的圆括号()实际上是C++中的操作符,它可以使函数被调用。在不采用参数函数调用中,忘记使用圆括号()并不是语法错误。但在你打算执行函数调用时,函数可能无法被调用。
- 3.9 在一个函数中定义另一个函数是语法错误。
- 3.10 如果形参和实参的函数原型、函数头部和函数调用在数目、类型、出现顺序以及返回值类型不一致,就表明这是语法错误。
- 3.11 忘记在函数原型末尾添加分号是语法错误。
- 3.12 不符合函数原型的调用是语法错误。
- 3.13 函数原型和函数定义不一致,也是语法错误。
- 3.14 提升规则中,把高级数据类型转换为低级数据类型会更改数据值。
- 3.15 函数未在初次调用之前定义,遗漏函数原型是语法错误。
- 3.16 用 srand 来代替 rand,令其生成随机数是语法错误,因为 srand 函数没有返回值。
- 3.17 为枚举类型的变量分配一个等同于枚举变量的整数值是语法错误。
- 3.18 枚举常量已被定义之后,再次尝试为其分配另一个值是语法错误。
- 3.19 针对一个标识符,使用多个存储类说明符是语法错误。一个标识符只能使用一个存储类说明符。例如,如果把一个标识符指定为 register,就不能再将其设为 auto。
- 3.20 如果出于偶然,对内部块和外部块中的标识符采用了同样的名称,而程序员事实上却希望外部块中的标识符在内部块存在期间处于活动状态,通常属于逻辑错误。
- 3.21 需要递归函数返回值时,如果忘记返回值将导致大多数编译器生成一条警告消息。
- 3.22 省略基本条件,或把递归步骤误写为不能到推回基本条件都可能导致“无穷递归”,最终耗尽内存。这如同在迭代(非递归)解决方法中的无限循环问题。导致无穷递归的另一种可能是意外的输入错误。
- 3.23 对于非 &&、||、逗号(,)和?:操作符的操作符操作数,如果要想写依赖于其求值顺序的程序,可能会出现错误,因为编译器也许不会按程序员期望的那样对操作数进行求值。
- 3.24 出于失误,令非递归函数通过另一个函数直接或间接调用其本身是逻辑错误。
- 3.25 C++ 程序不会得以编译,除非提供每个函数的函数原型或在调用函数之前定义每个函数。
- 3.26 由于在被调用函数体中,引用参数只能通过名称未指定,所以程序员可能会因此把引用参数视为传值调用参数。这样一来,如果变量的原始副本已被调用函数修改,就会导致不可预测的副作用。

- 3.27 在用圆点分隔的变量名称列表中,多处使用 & 的同时,要在语句中声明多个引用。要想把变量 x, y 和 z 全部声明为整数引用时,应该使用 `int &x = a, &y = c, &z = c` 这 3 个表达式,而不应该使用不正确的表这式 `int & x = a, y = b, z = c`,或者其他常见的错误表达式 `int &x, y, z`。
- 3.28 声明引用变量时未将其初始化是语法错误。
- 3.29 试图将一个已声明的引用重新指定为另一个变量的别名是逻辑错误。另一个变量的值仅被分配给已作为别名的引用所在的地址。
- 3.30 向被调用函数中的自动变量返回指针或引用是逻辑错误。对此,有的编译器会发出警告消息。
- 3.31 指定并打算使用非最右边(追尾的)实参的默认实参(也就没有把所有最右边的实参指定为默认实参)是语法错误。
- 3.32 试图利用一元作用域分辨符来访问外层块中的非全局变量时,如果外层块中没有与该变量同名的全局变量,就会出现语法错误,反之,如果有与该变量同名的全局变量,则会出现逻辑错误。
- 3.33 创建参数列表相同,而返回类型不同的重载函数是语法错误。
- 3.34 像调用另一个重载函数那样调用省略默认参数的函数会产生语法错误。例如在程序中,如果同时包含显式不取参数的函数和另一个其中带有所有默认参数的同名函数,那么打算在没有传递任何参数的调用中使用这个函数名就会产生语法错误。
- 3.35 在函数模板的每个类型参数前,不加上关键字 `class` 或 `typename` 是语法错误。

## 良好编程习惯

- 3.1 尽快掌握 C++ 标准库中的函数和类集合。
- 3.2 在函数定义之间放置一个空行,以分隔函数并增强程序的可读性。
- 3.3 尽管不会错,但最好不要使传递给函数的形参与函数定义中的实参同名。这样可以避免歧义性。
- 3.4 选择含义明确的函数名称和参数名可以使程序更具可读性,而且还有助于避免使用大量的注释。
- 3.5 许多程序员使用函数原型中的参数名来描述函数。编译器会忽略这些名称。
- 3.6 使用用户自定义类型名称的标识符,其首字母应该大写。
- 3.7 在枚举常量中,只采用大写字母。这样可以使这些常量在程序中更能引起程序员注意,而且,还可以提醒程序员枚举常量并不是变量。
- 3.8 在程序中,使用枚举来代替整数常量可以使程序更清晰。
- 3.9 避免隐藏外部块范围的变量名。这是通过避免在程序中使用重复性的标识符来实现的。
- 3.10 始终记得提供函数原型,即使函数在使用之前的定义时可能被省略。提供函数原型可以避免代码按函数定义的顺序出现(可轻易随程序的改变而改变)。
- 3.11 `inline` 限定符应该只适用于小型的、使用较为频繁的函数。
- 3.12 使用默认实参可以简化函数调用的编程工作。但也有的程序员认为,显式指定所有实

参会使程序更清晰。

- 3.13 尽量避免在程序中出现用途不同的同名变量。尽管在许多情形下允许这样做,但这样往往会产生混乱。
- 3.14 执行类似任务的函数重载功能可以令程序更易于阅读和理解。

### 性能提示

- 3.1 对于现有的库程序,不要为了使其更有效而试图重写它。这些程序的性能通常已被发挥得淋漓尽致,没有提升的余地。
- 3.2 `srand` 函数只需在程序中调用一次,就可以获得所需的随机化结果。多次调用不仅徒劳无益,还会降低程序性能。
- 3.3 自动存储意味着可以节省内存,因为自动存储类变量会在进入声明的块时创建,并在退出块时删除。
- 3.4 存储类说明符 `register` 可以置于自动变量声明之前,以使编译器在计算机的高速硬件寄存器中而不是在内存中维护该变量。如果可以在硬件寄存器中维护频繁使用的变量(比如计数器、总和),那么可以避免重复将变量从内存装入寄存器以及将结果返回内存。
- 3.5 通常情况下,`register` 声明不是必不可少的。如今的优化编译器能识别使用较为频繁的变量,并能在程序员不提供 `register` 声明的情况下,将这些变量置于寄存器中。
- 3.6 避免使用会造成调用呈指数级递增的 fibonacci 式递归程序。
- 3.7 避免在对性能要求较高的情形下使用递归。递归调用既费时又占用较多的内存。
- 3.8 一个由多个函数组成的程序——与没有任何函数的一体式程序相比——会产生大量的函数调用,而且这些调用会占用执行时间和计算机处理器的空间。但一体式程序的编程、测试、调试、维护和改进都比较复杂。
- 3.9 使用内联函数会减少执行时间,但是要增加程序的长度。
- 3.10 传值调用的缺点是,如果传递的是大型数据项目,复制该数据就会花较长的时间。
- 3.11 引用调用对性能有一定好处,因为它可以避免复制大量数据。
- 3.12 对于传递大型对象,可用常量引用参数来模拟传值调用的外观和安全性,进而避免传递大型对象副本。

### 可移植性提示

- 3.1 利用 C++ 标准库中的函数有助于提高程序的可移植性。
- 3.2 依赖于非 `&&`、`||`、逗号 `,` 和 `?:` 操作符操作数的程序在系统中的作用会因为编译器的不同而不同。
- 3.3 C++ 中,空函数参数列表的含义与 C 中有显著差别。在 C 语言中,它意味着所有参数检查都是无效的(也就是说,函数调用可以传递任何它想传递的参数)。在 C++ 语言中,则意味着函数没有参数。因此,在 C++ 中编译采用该特性的 C 程序时,可能会报告语法错误。



## 软件工程知识

- 3.1 避免万事从头开始。在可能的情况下,尽量使用C++标准库中的函数,而不是从头编写新函数。这样可以减少程序开发时间。
- 3.2 在包含许多函数的程序中,main应被视作一组函数调用来执行,这组函数用于执行程序的大部分工作。
- 3.3 每个函数都应限于只完成一个单独的、具有良好定义的任务,而且函数名称应该更能有效地代表它所完成的任务。这样可以提升软件的重用性。
- 3.4 如果实在无法想出能准确表达函数作用的名称,则表明你定义的函数执行的任务可能太分散。碰到类似情况,最好把这类函数分割成几个较小的函数,令其各自负责某项特定任务。
- 3.5 函数应该适应编辑器窗口的大小。不管函数有多长,它都应该能很好地完成任务。小函数有助于提升软件的重用性。
- 3.6 程序应该写成若干小函数的集合。这样可以使程序更易于编写、调试、维护和修改。
- 3.7 需要大量参数的函数可能会执行大量任务。此时,可考虑把函数分解成更小的函数,令各个小函数分担个别任务。必要的情况下,函数的头部应该包含在单独的一行内。
- 3.8 C++语言中,函数原型是必须的。利用#include预处理程序指令可从相应库的头文件中获得标准库函数的函数原型。此外,使用#include还可获得其中包含你和小组成员所用函数原型的头文件。
- 3.9 如果函数定义出现在程序第一次使用该函数之前,就不需要函数原型。此时,函数的定义就充当了函数原型。
- 3.10 在文件中,放置于任何函数定义以外的函数原型适用于出现在函数原型之后的所有对该函数的调用。放置于函数内部的函数原型只适用于该函数内部执行的调用。
- 3.11 自动存储是最低权限级的实例。不需要变量时,为何要将其保存在内存中供存取呢?
- 3.12 如果把变量声明为全局变量而非局部变量,那么对于不需要访问变量的函数而言,如果它有意或无意修改该变量时,可能引起难以预料的副作用。一般说来,除非有特殊的性能需求,多数情况下都应尽量避免使用全局变量。
- 3.13 只用于特定函数的变量应该被声明为该函数的局部变量而非全局变量。
- 3.14 可以采用递归方式处理的任何问题也可采用迭代方式(非递归方式)解决。递归法更能反映问题并令程序易于理解和调试,递归法通常优于迭代法。选择递归法的另一个理由是迭代法不直观。
- 3.15 令程序以清晰的层次化结构运行,可提升软件工程的质量,但要付出一定的代价。
- 3.16 对内联函数的任何更改都需要重新编译该函数的所有客户。这会大大影响某些程序的开发和维护。
- 3.17 许多程序员不会多此一举地把数值参数声明为const,即使被调用的函数不会修改所传递的参数(实参)。使用关键字const的目的只是为了保护原始实参的副本,而不是原始实参本身。
- 3.18 引用调用的安全性较差,因为被调用函数会直接访问并修改调用者的数据。

- 3.19 综合考虑程序的清晰性和高性能,许多C++程序员偏向于使用指针,把可修改的参数传递给函数,小型的非修改性参数可以传值调用,大型的非修改参数则可以利用常量引用传递给函数。

### 测试和调试提示

- 3.1 即使能完全确定程序中没有错误,也应在 switch 结构中用 default case 来捕捉错误。

### 自测题

#### 3.1 填空题:

- a) C++ 中的程序组件称为\_\_\_\_\_和\_\_\_\_\_。
- b) 函数的执行是通过\_\_\_\_\_来实现的。
- c) 对一个变量而言,如果只能在定义了它的函数内部访问,那么这种变量就是\_\_\_\_\_。
- d) 被调用函数中,\_\_\_\_\_语句用于把表达式的值传回调用函数。
- e) 关键字\_\_\_\_\_用于函数头部,表明函数没有返回值或表明函数不取任何参数。
- f) 标识符的\_\_\_\_\_是指标识符可以用于程序中的这个部分。
- g) 控制权从被调用函数返回调用者的方式有 3 种,分别\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- h) \_\_\_\_\_允许编译器检查传递给函数的参数个数、类型和顺序。
- i) 函数\_\_\_\_\_用于生成随机数。
- j) 函数\_\_\_\_\_用于设置随机化种子数,以便实现程序的随机化。
- k) 存储类说明符是 mutabel、\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- l) 如果程序员不指定,程序块中声明的变量和函数参数列表中的变量就会被设置为存储类\_\_\_\_\_。
- m) 存储类说明符\_\_\_\_\_用于建议编译器把变量保存在计算机的高速寄存器中。
- n) 任何一个程序块或函数外部声明的变量是一个\_\_\_\_\_变量。
- o) 对于函数中的局部变量,如果要令其值在函数调用期间维持原值,就必须用\_\_\_\_\_存储类说明符来声明。
- p) 标识符的 4 个作用域分别是\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- q) 直接或间接调用其自身的函数是一个\_\_\_\_\_函数。
- r) 递归函数一般有两个组件:一个提供了一种方式,即通过对\_\_\_\_\_的测试来中止递归调用,另一个则是将原问题简化为小问题。
- s) 在C++中,可以有多个名称相同,但参数类型不同和/个数不同的函数。这种方式叫作函数\_\_\_\_\_。
- t) \_\_\_\_\_可以访问变量当前范围内的同名全局变量。
- u) \_\_\_\_\_限定符用于声明只读变量。
- v) 函数\_\_\_\_\_可以把一个函数定义为对不同数据类型执行相同的任务。

- 3.2 查看下列程序,指出以下各元素的作用域(包括函数范围、文件范围、程序块范围和函数

原型范围):

- a) main 中的变量 x。
- b) cube 中的变量 y。
- c) 函数 cube。
- d) 函数 main。
- e) cube 的函数原型。
- f) cube 函数原型中的标识符 y。

```
1 //ex03_02.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::endl;
6
7 int cube(int y);
8
9 int main()
10 {
11     int x;
12
13     for (x = 1; x <= 10; x++)
14         cout << cube(x) << endl;
15
16     return 0;
17 }
18
19 int cube(int y)
20 {
21     return y*y*y;
22 }
```

3.3 编写一个程序,测试图 3.2 中的代数学函数调用示例是否真的能生成图中所示结果。

```
1 //ex03_02.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::endl;
6
7 int cube(int y);
8
9 int main()
10 {
11     int x;
12
13     for (x = 1; x <= 10; x++)
14         cout << cube(x) << endl;
15
16     return 0;
17 }
```

```

18
19 int cube( int y )
20 |
21     return y * y * y;
22 |

```

3.4 指出以下各个函数的函数头部:

- 函数 `hypotenuse` 采用了两个双精度的浮点参数 `side1` 和 `side2`, 并返回了一个双精度的浮点结果值。
- 函数 `smallest` 采用了 3 个整数值参数, 并返回一个整数。
- 函数 `instructions` 不接收任何参数, 也不返回任何值(注意, 此类函数常用于向用户显示指令)。
- 函数 `intToDouble` 采用了一个整数参数 `number`, 并返回一个单精度浮点结果值。

3.5 指出以下各函数的函数原型:

- 练习题 3.4a) 中的函数。
- 练习题 3.4b) 中的函数。
- 练习题 3.4c) 中的函数。
- 练习题 3.4d) 中的函数。

3.6 为以下各数编写声明:

- 将保存在寄存器中的整数 `count`。把 `count` 初始化为 0。
- 双精度浮点变量 `lastVal`。其值在定义了该数值的函数调用期间保持不变。
- 外部整数 `number`, 其范围被限制在定义了该整数的文件以内。

3.7 指出以下各程序片段中的错误, 并说明如何改正(也可参见练习题 3.53):

```

a) int g(void){
    cout << "Inside function g" << endl;

    int h(void){
        {
            cout << "Inside function h" << endl;
        }
    }
}

b) int sum(int x, int y)
|
    int result;

    result = x + y;
|

c) int sum(int n)
|
    if (n == 0)
        return 0;
    else
        n + sum(n - 1)

```

```

    }
d) void f(double a);
    |
    float a;
    cout <<a<<endl;
    |
e) void product (void)
    |
    int a, b, c, result;
    cout << "Enter three integers:";
    cin >>a>>b>>c;
    result = a*b*c;
    cout << "Result is" << result;
    return result;
    |

```

3.8 为什么函数原型中要包含 double & 之类的参数类型声明?

3.9 (判断正误) C++ 中, 所有调用都是传值调用。

3.10 编写一个完整的程序, 使其利用一个内联函数 sphereVolume 来提醒用户输入球半径, 并通过公式  $\text{volume} = (4.0/3) * 3.14159 * \text{pow}(\text{radius}, 3)$ , 计算并打印球的体积。

### 自测题答案

- 3.1 a) 函数和类 b) 函数调用 c) 局部变量 d) return e) void f) 作用域  
 g) return; 或 return expression; 或碰到标志函数结束的右花括号时 h) 函数原型  
 i) rand j) srand k) auto, register, extern, static l) auto m) register  
 n) 外部、局部 o) static p) 函数范围、文件范围、程序块范围、函数原型范围  
 q) 递归 r) 基本 s) 重载 t) 一元作用域分辨符(::) u) const v) 模板
- 3.2 a) 程序块范围 b) 程序块范围 c) 文件范围 d) 文件范围  
 e) 文件范围 f) 函数原型范围
- 3.3 如下所示:

```

1 //ex03_03.cpp
2 //Testing the math library func
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8
9 #include <iomanip>
10
11 using std::setiosflags;
12 using std::fixed;
13 using std::setprecision;
14
15 #include <cmath>

```

```

16
17 int main()
18 {
19     cout << setiosflags( ios::fixed | ios::showpoint )
20         << setprecision( 1 )
21         << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
22         << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 )
23         << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
24         << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
25         << ") = " << setprecision( 6 ) << exp( 2.0 )
26         << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
27         << log( 2.718282 ) << "\nlog(" << setprecision( 6 )
28         << 7.389056 << ") = " << setprecision( 1 )
29         << log( 7.389056 ) << endl;
30     cout << "log10(" << 1.0 << ") = " << log10( 1.0 )
31         << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
32         << "\nlog10(" << 100.0 << ") = " << log10( 100.0 )
33         << "\nfabs(" << 13.5 << ") = " << fabs( 13.5 )
34         << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
35         << "\nfabs(" << -13.5 << ") = " << fabs( -13.5 ) << endl;
36     cout << "ceil(" << 9.2 << ") = " << ceil( 9.2 )
37         << "\nceil(" << -9.8 << ") = " << ceil( -9.8 )
38         << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
39         << "\nfloor(" << -9.8 << ") = " << floor( -9.8 ) << endl;
40     cout << "pow(" << 2.0 << ", " << 7.0 << ") = "
41         << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
42         << 0.5 << ") = " << pow( 9.0, 0.5 )
43         << setprecision(3) << "\nfmod("
44         << 13.675 << ", " << 2.333 << ") = "
45         << fmod( 13.675, 2.333 ) << setprecision( 1 )
46         << "\nsin(" << 0.0 << ") = " << sin( 0.0 )
47         << "\ncos(" << 0.0 << ") = " << cos( 0.0 )
48         << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
49     return 0;
50 }

```

输出结果:

```

sqrt(900.0)=30.0
sqrt(9.0)=3.0
exp(1.0)=2.718282
exp(2.0)=7.389056
log(2.718282)=1.0
log(7.389056)=2.0
log10(1.0)=0.0
log10(10.0)=2.0
log10(100.0)=2.0
fabs(13.5)=13.5
fabs(0.0)=0.0
fabs(-13.5)=13.5
ceil(9.2)=10.0
ceil(-9.8)=-9.0

```

```

floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

- 3.4 a) `double hypotenuse(double side1, double side2)`  
 b) `int smallest(int x, int y, int z)`  
 c) `void instructions(void)` //in C++ (void) can be written ()  
 d) `float intToDouble(int number)`

- 3.5 a) `double hypotenuse(double, double);`  
 b) `int smallest(int, int, int);`  
 c) `void instructions(void);` //in C++ (void) can be written()  
 d) `float intToDouble(int);`

- 3.6 a) `register int count = 0;`  
 b) `static double lastVal;`  
 c) `static int number;`

说明: 声明将出现在任何函数定义外部。

- 3.7 a) 错误: 函数 h 的定义出现在函数 g 中。

改正: 把函数 h 移到函数 g 外部定义。

- b) 错误: 函数应该返回整数, 实际上却没有。

改正: 删除 result 变量, 并在函数中添加以下语句:

```
return x + y;
```

- c) 错误: 没有返回 `n + sum(n - 1)` 的结果; sum 返回的结果不正确。

改正: 在 else 语句中重写语句, 如下所示:

```
return n + sum(n - 1);
```

- d) 错误: 用于包括参数列表的右圆括号之后, 出现了分号, 而且对函数定义中的 a 进行了重新定义。

改正: 删除参数列表的右圆括号, 删除 `float a;` 声明。

- e) 错误: 不该有返回值时, 函数却返回了值。

改正: 删除 return 语句。

- 3.8 因为程序员声明了 double 引用类型的引用参数, 打算通过引用调用访问原始参数变量。

- 3.9 错。除了使用指针外, C++ 还允许通过引用参数的使用来实现直接引用调用。

- 3.10 答案如下:

```

1 //ex03_10.cpp
2 //Inline function that calculates the volume of a sphere
3 #include <iostream>
4
5 using std::cout;

```

```

6  using std::cin;
7  using std::endl;
8
9  const double PI = 3.14159;
10
11 inline double sphereVolume( const double r )
12     | return 4.0 /3.0 * PI * r * r * r; {
13
14 int main()
15 |
16     double radius;
17
18     cout << "Enter the length of the radius of your sphere: ";
19     cin >> radius;
20     cout << "Volume of sphere with radius " << radius <<
21         "is" << sphereVolume( radius ) << endl;
22     return 0;
23 |

```

### 练习题

3.11 执行以下语句之后,显示 x 的值:

- a) `x = fabs(7.5)`
- b) `x = floor(7.5)`
- c) `x = fabs(0.0)`
- d) `x = ceil(0.0)`
- e) `x = fabs(-6.4)`
- f) `x = ceil(-6.4)`
- g) `x = ceil(-fabs(-8 + floor(-5.5)))`

3.12 停车场的收费方案是3小时以内至少收取2美元的停车费。超过3小时,每增加一小时,加收0.5美元,24小时的最高收费为10美元。假设任何车辆的停车时间都不超过24小时。请根据这一情况编写一个程序,计算并打印昨天在该停车场停车的3个顾客的停车费。你应输入每个顾客的停车时间。程序应以整齐的表格形式打印结果,并计算和打印昨天收到的所有费用的总和。程序应用 `calculateCaharges` 函数来判断每个顾客的停车费。程序的输出结果如下所示:

Car	Hours	Charge
1	1.5	2.00
2	4.0	2.50
3	24.0	10.00
TOTAL	29.5	14.50

3.13 函数 `floor` 的一种用法是对一个数取整为其最接近的整数。语句

```
y = floor( x + .5 );
```

将 x 取整为最接近于它的整数,并将结果赋予 y。编写一个程序,令其读取几个数值,并用前面的语句对这几个数值取整为最接近的整数。针对处理后的每个数值,打印其



原始数值及其取整后的数值。

- 3.14 函数 `floor` 可用于将数值取整为特定的小数位。语句

```
y = floor( x * 10 + .5 ) / 10;
```

将 `x` 取整为十分位(小数点右边的第一位)。语句

```
y = floor( x * 100 + .5 ) / 100;
```

将 `x` 取整为百分位(小数点右边的第二位)。编写一个程序,定义 4 个函数,令其用不同的方式对 `x` 取整:

- a) `roundToInteger( number )`
- b) `roundToTenths( number )`
- c) `roundToHundredths( number )`
- d) `roundToThousandths( number )`

对于读取的每一个值,程序都将打印出原始数值,取整为最接近整数后的结果值、取整为十分位后的结果值、取整为百分位后的结果值以及取整为千分位后的结果值。

- 3.15 回答以下问题:

- a) 何为“随机”选择数值?
- b) 为什么 `rand` 函数对模拟机会游戏特别有用?
- c) 为什么要用 `srand` 来实现程序的“随机化”? 什么情况下不宜使用随机化?
- d) 为什么通常需要按比例缩放和/或移位 `rand` 生成的值?
- e) 为什么说用计算机来模拟真实世界是个非常有用的技巧?

- 3.16 编写语句,把下列范围内的随机整数分配给变量 `n`:

- a)  $1 \leq n \leq 2$
- b)  $1 \leq n \leq 100$
- c)  $0 \leq n \leq 9$
- d)  $1\,000 \leq n \leq 1\,112$
- e)  $-1 \leq n \leq 1$
- f)  $-3 \leq n \leq 11$

- 3.17 针对以下各组整数,编写一条语句,随机打印出它们。

- a) 2,4,6,8,10
- b) 3,5,7,9,11
- c) 6,10,14,18,22

- 3.18 编写一个函数 `integerPower( base, exponent )`,该函数的返回值是

$\text{base}^{\text{exponent}}$

例如, `integerPower(3,4) = 3 * 3 * 3 * 3`。假设 `exponent` 是个正数,即非零的整数, `base` 是个整数。函数 `integerPower` 会用 `for` 或 `while` 来控制计算。不要用任何一个代数库函数。

- 3.19 定义函数 `hypotenuse`,令其计算已知直角三角形其两边边长时第三边的边长。在程序中利用该函数来判断以下各三角形的边长。函数将取两个 `double` 类型的参数,并返回 `double` 类型的结果值。

Triangle	Side1	Side2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

- 3.20 编写一个 `multiple` 函数,令其判断一对整数中,第二个整数是否是第一个整数的倍数。函数要取两个整数参数,并在第二个整数是第一个整数的倍数时返回 `true`,反之则返回 `false`。在程序中利用该函数,并输入一系列整数对。
- 3.21 编写一个程序,在其中输入一系列整数,并将它们传递给函数 `even`,一次传递一个,程序中将用求模操作符来判断整数是否为偶数。函数应取一个整数参数,并在整数是偶数时返回 `true`,反之则返回 `false`。
- 3.22 编写一个函数,令其在屏幕左边显示一个星形的实心正方形,该正方形的边长是在整数参数 `side` 中指定的。例如,如果 `side` 是 4,函数就会显示:

```
* * * *
* * * *
* * * *
* * * *
```

- 3.23 修改练习题 3.22 中创建的函数,用字符参数 `fillCharacter` 中的字符来填充正方形。因此,如果 `side` 是 5, `fillCharacter` 是“#”,那么该函数就会显示

```
#####
#####
#####
#####
#####
```

- 3.24 利用练习题 3.22 和 3.23 中的类似技巧,生成一个程序,使之能绘制各种图形。
- 3.25 编写程序片段,分别完成以下任务:
- 整数 `a` 除以整数 `b` 时,计算其商的整数部分。
  - 整数 `a` 除以整数 `b` 时,计算其商的整数余数。
  - 利用 a) 和 b) 编写的程序片段编写一个程序,输入 1 到 32 767 之间的整数值,并将其作为一系列数字打印出来,各对数字之间用两个空格分开。例如整数 4 562 被打印为
- ```
4 5 6 2
```
- 3.26 编写一个函数,将时间作为 3 个整数参数(分别代表时、分和秒),并返回自上一次时钟的“12 点整”算起的秒数。利用该函数来计算两个时间之间的秒数,两者都不得超过 12 小时。
- 3.27 实现下列整数函数:
- 函数 `celsius` 返回华氏温度对应的摄氏温度。
  - 函数 `fahrenheit` 返回摄氏温度对应的华氏温度。
  - 利用这两个函数编写一个程序,用图表的形式打印从 0 ~ 100 度之间的所有摄氏温度对应的华氏温度,32 ~ 212 度之间所有华氏温度对应的摄氏温度。以整齐的表格形式打印输出结果,并在保证可读性的情况下,减少输出行数。
- 3.28 编写一个函数,令其返回 3 个双精度浮点数中的最小值。

- 3.29 对于一个整数,如果其因子的总和,包括 1 在内,(但不包括该整数本身)等于该整数时,我们就可以把该整数称为“完数”(perfect number)。例如 6 就是一个完数。因为  $6 = 1 + 2 + 3$ 。编写一个函数 perfect,判断 number 参数是否为一个完数。在程序中利用该函数来判断并打印 1 到 1 000 之间的所有完数。打印每个完数的因子,确定它们的确是完数。测试大于 1 000 的数,向计算机的计算能力发出挑战。
- 3.30 对于一个整数,如果它只能被 1 和其本身整除,那么这个数就是“质数”(prime)。例如,2,3,5 和 7 是质数,而 4,6,8,9 则不是。
- 编写一个函数,判断一个数是否为质数。
  - 在程序中使用该函数,判断并打印 1 到 10 000 之间的所有质数,以及在确定这 10 000 个数中,需要测试多少个数,才能找出所有的质数?
  - 首先,你会认为  $n/2$  是确定一个数是否为质数的上限,但事实上,只需要  $n$  的平方根次。为什么? 改写程序,用这两种方式来运行。估计一下系统性能改进了多少。
- 3.31 编写一个函数,取一个整数值,返回其数值取反序之后的结果值。例如,输入 7 631,函数就会返回 1 367。
- 3.32 两个整数的最大公约数(greatest common divisor,简称 GCD)是这两个整数能够整除的最大整数。编写一个 gcd 函数,返回两个整数的最大公约数。
- 3.33 编写一个 qualityPoints 函数,输入学生的平均成绩,并在学生平均成绩为 90 ~ 100 时返回 4,为 80 ~ 89 时返回 3,为 70 ~ 79 时返回 2,为 60 ~ 69 时返回 1,低于 60 时返回 0。
- 3.34 编写一个程序模拟投币游戏。每次投币,程序都会打印 Heads(正面)或 Tails(反面)。让程序模拟 100 次投币动作,并计算硬币各面出现的次数。打印结果。程序会调用一个单独的函数 flip,该函数不取任何参数,便在出现硬币正面时返回 0,在出现反面时返回 1。注意:如果程序按实际情况模拟投币游戏,那么硬币正反面出现的机会会相等。
- 3.35 计算机在教育领域内扮演着越来越重要的角色。编写一个有助于小学生学习乘法的程序。利用 rand 函数生成两个一位的整数。随后,它会输入如下所示的问题:

How much is 6 times 7?

然后由学生输入答案。你编写的程序用于检查学生们的答案。如果答案正确,就打印“Very good!”,接着进行下一次运算。如果答案错误,就打印“No, Please try again.”让学生再次输入答案直到答对为止。

- 3.36 计算机在教育领域内的使用也就是人们常说的“计算机辅助教学(CAI)”。在 CAI 环境中容易出现的问题是学生容易疲劳。不过,这是可以消除的,具体做法是通过计算机对话框的多样化来让学生保持注意力。修改练习题 3.35 中的程序,像下面这样,为学生的每次正确或错误答案打印不同的评语:

答对时打印:

Very good!  
Excellent!  
Nice work!  
Keep up the good work!

答错时打印:

No. Please try again.

```
Wrong, Try once more.
Don't give up!
No, Keep trying.
```

利用随机数生成器,在 1~4 之间选择一个数,并因此为学生的每个答案选择相应的评语。利用 switch 结构发出响应。

- 3.37 更复杂的计算机辅助教学系统会对学生一段时间的表现进行监视。开始新主题的决定往往是在前期已获成功的主题基础上得出的。修改练习 3.36 中的程序,计算学生输入的正确答案和错误答案的次数。学生输入 10 个答案之后,你的程序会计算其答对率。如果答对率低于 75%,程序就会打印出“Please ask your instructor for extra help”并结束。
- 3.38 像这样编写一个猜数字游戏程序:程序在 1~1000 之间随机选择一个数字作为答案。然后输入

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

然后,游戏者输入猜想的第一个数字。程序会作出以下应答

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

如果游戏者的猜测是错的,程序就会执行循环,直到游戏者最终猜对为止。你的程序要一直提示游戏者 Too high 或 Too low 帮助他们尽快获得正确答案。

- 3.39 修改练习题 3.38 中的程序,并计算游戏者猜测的次数。如果该数为 10 或更少,就打印“Either you know the secret or you got lucky!”。如果游戏者猜测次数为 10,就打印“Ahah! You know the secret!”。如果游戏者猜测次数超过 10 次,就打印“You should be able to do better!”。为什么猜测次数不能超过 10 次呢?是这样的,作为真正的猜数字游戏高手,应该 5 次以内就能猜出正确答案。现在,请展示如何在 10 次或更少的次数内猜出 1 到 1 000 之间的正确数字。
- 3.40 编写一个递归函数  $\text{power}(\text{base}, \text{exponent})$ ,调用该函数时,返回结果为

$$\text{base}^{\text{exponent}}$$

例如,  $\text{power}(3,4) = 3 * 3 * 3 * 3$ 。前提是 exponent 是个大于或等于 1 的整数。提示:递归步骤会使用以下关系

$$\text{base}^{\text{exponent}} = \text{base} * \text{base}^{\text{exponent}-1}$$

exponent 等于 1 时停止递归,因为

$$\text{base}^1 = \text{base}$$

- 3.41 费波拉奇数列

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

以 0 和 1 开头,后面的数是前两个数的和。a) 编写一个非递归函数  $\text{fibonacci}(n)$ , 计算第  $n$  个费波拉奇数。b) 确定自己的系统能打印最大的费波拉奇数。修改 a) 部分的程序,用 double 类型值来代替 int 类型值,计算并返回费波拉奇数,并用修改后的程序重

复执行 b) 部分的任务。

- 3.42 (汉诺塔问题) 每个年轻的计算机科学家都必须掌握一些特定的经典问题。汉诺塔(如图 3.33 所示)便是最著名的问题之一。传说中称, 远东地区的一个寺庙内, 有一群僧人, 打算把一堆碟子从一个柱子挪到另一个柱子。原先那个柱子共有 64 个碟子, 这 64 个碟子随着尺寸的减小, 从下到上串成一串。僧人们打算把这些碟子从一个柱子挪到另一个柱子, 前提是一次只挪一个碟子, 而且大碟子不能放在小碟子之上。第三个柱子可用于临时存放碟子。也就是说, 僧人们完成任务之时, 也就是世界末日到来之时, 所以, 我们还是悠着点, 别让这一时刻提前到来。

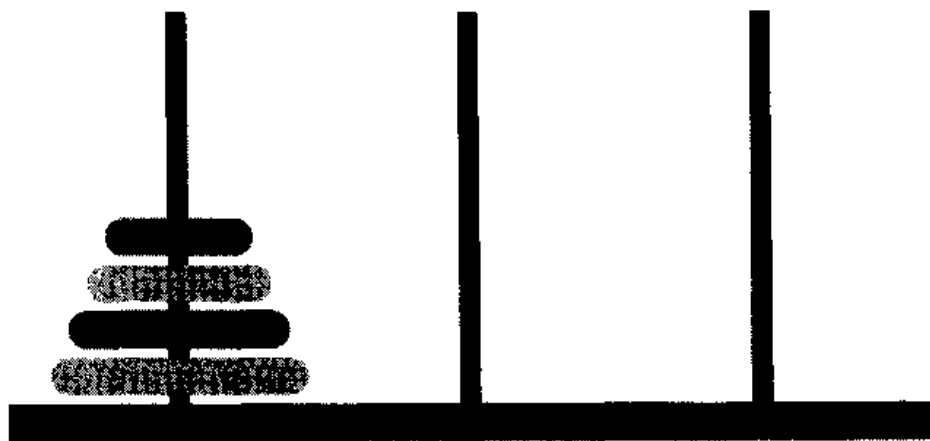


图 3.33 4 个碟子的汉诺塔问题

我们假设这些僧人打算把碟子从第 1 个柱子挪到第 3 个柱子。我们希望开发出一种算法, 打印出挪动碟子的简略顺序。

如果用常规方法解决这个问题, 立刻就会感觉到它的复杂性非同一般。相反地, 如果我们用递归方法, 就可以理清脉络。移动  $n$  个碟子就变成移动  $n-1$  个碟子(这就是递归的妙用), 如下所示:

- a) 把  $n-1$  个碟子从第一个柱子挪到第二个柱子, 将第三个柱子用作临时存放点。
- b) 把位于底层的那个大碟子(最大的那个)从第一个柱子挪到第 3 个柱子。
- c) 把  $n-1$  个碟子从第二个柱子挪到第 3 个柱子, 将第一个柱子用作临时存放点。

整个过程将在执行最后一个任务: 移动  $n=1$  个碟子, 也就是碰到基本情况时结束。这时, 无需临时存放点也可以轻松移动最后一个碟子。

编写一个程序来解决汉诺塔问题。利用带有以下 4 个参数的递归函数:

- a) 准备移动的碟子的数量。
- b) 最初存放这些碟子的柱子。
- c) 最后存放这些碟子的柱子。
- d) 作为临时存放点的柱子。

你的程序会打印把这些碟子从原主子到目标柱子的简要步骤。例如, 要想把 3 个碟子从第一个柱子挪到第 3 个柱子, 程序就会如下所示的步骤:

1→3(这表示把一个碟子从第一个柱子挪到第 3 个柱子)

1→2

3→2

1→3

2→1

2→3

1→3

- 3.43 任何可以通过递归方式实现的程序都可以用迭代的方式来实现,尽管有时会增加难度和减弱清晰程度。试着写一个程序,用迭代方式来解决汉诺塔问题。如果成功,就比较一下迭代程序和练习题 3.42 中的递归程序。注意程序的性能、清晰程度以及自己演示程序正确性的能力。
- 3.44 (图形化递归)随时查看递归的实现过程是非常有趣的。修改图 3.14 中的阶乘函数,打印其局部变量和递归调用参数。针对每个递归调用,在一行内显示其输出结果,并增加缩进量。尽量使输出结果清晰、有趣而具有一定意义。你的目的是设计和实现一种输出格式,能让读者更好地理解递归。你还可以为本书的其他递归范例和练习添加类似的显示效果。
- 3.45 整数  $x$  和  $y$  的最大公约数是  $x$  和  $y$  都能整除的最大整数。编写一个递归函数 `gcd`,令其返回  $x$  和  $y$  的最大公约数。 $x$  和  $y$  的 `gcd` 函数的递归定义如下:如果  $y$  等于 0,那么 `gcd(x,y)` 就等于  $x$ ;否则 `gcd(x,y)` 就等于 `gcd(y,x%y)`,这里的 `%` 是求模操作符。
- 3.46 可以递归调用 `main` 函数吗?编写一个其中包含 `main` 的程序。要包含静态局部变量 `count`,并将其初始化为 1。每次调用 `main` 时,要先递增 `count`,并打印其值。然后编译并运行程序,观察结果。
- 3.47 练习题 3.35 和练习题 3.37 开发了一个计算机辅助教学程序帮助小学生学习乘法。本练习建议大家改进这个这个程序。
- 修改程序,让用户能输入级别。我们指的级别只能用一个数位来表示问题。比如,2 级表示我们最多用 2 位数的数字。
  - 修改程序,让用户能选择他或她打算学习的算术问题的类型。选项 1 代表加法问题,选项 2 代表减法问题,选项 3 代表乘法问题,选项 4 代表除法问题,选项 5 代表随机的混合运算。
- 3.48 编写函数 `distance`,计算两个点  $(x_1, y_1)$  和  $(x_2, y_2)$  之间的距离。所有数字和返回结果都应为 `double` 类型。
- 3.49 指出下列程序中的错误。

```

1  //ex03_49.cpp
2  #include <iostream>
3
4  using std::cin;
5  using std::cout;
6
7  int main()
8  |
9      int c;
```

```

10
11     if ((c = cin.get()) != EOF) |
12         main();
13         cout << c;
14     |
15
16     return 0;
17 }

```

### 3.50 指出下列程序的用途。

```

1 //ex03_50.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int mystery(int,int);
9
10 int main()
11 |
12     int x,y;
13
14     cout << "Enter two integers:";
15     cin >> x >> y;
16     cout << "The result is" << mystery(x,y) << endl;
17     return 0;
18 |
19
20 //parameter b must be a positive
21 //integer to prevent infinite recursion
22 int mystery (int a, int b)
23 {
24     if (b=1)
25         return a;
26     else
27         return a+mystery(a,b-1);
28 }

```

- 3.51 得知练习题 3.50 中程序的用途后,修改该程序,保证在删除第二个参数须为非负值的限制条件该程序仍然能正常工作。
- 3.52 编写一个程序,尽量测试图 3.2 中的大多数数学库函数。令自己编写的程序打印出各种参数值的对应返回值表,以达到逐一操练每个函数的目的。
- 3.53 找出以下程序段中的错误,并说明如何改正:

```

a) float cube (float);           //function prototype
   double cube (float number) //function definition
   {
       return number * number * number;
   }

```

```

    |
b) register auto int x=7;
c) int randomNumber = srand();
d) float y=123.45678;
    int x;
    x=y;
    cout << static_cast < float > (x) << endl;
e) double square (double number)
    |
    double number;
    return number * number;
    |
f) int sum (int n)
    {
        if (n==0)
            return 0;
        else
            return n + sum (n);
    }
    |

```

- 3.54 修改图 3.10 中的掷骰子程序,加入新的赌注。把程序中运行掷骰子的部分打包为一个函数。把变量 `bankBalance` 初始化为 1 000 美元。提示游戏者输入一个 `wager` (赌注)。利用 `while` 循环见 `wager` 是小于还是等于 `bankBalance`,如果不是就提示用户重新输入 `wager`,直到用户输入一个正确的 `wager` 为止。输入正确的 `wager` 之后,运行掷骰子游戏。如果游戏者赢了,就在 `bankBalance` 的基础上增加 `wager`,并打印出新的 `bankBalance`。如果游戏者输了,就在 `bankBalance` 的基础上减少 `wager`,然后打印出新的 `bankBalance`,检查 `bankBalance` 是否已变成 0,如果是,就打印这样的消息“Sorry. You busted!”。随着游戏的继续,还会打印别的消息,获得聊天效果,比如“*Oh, you're going fro broke, huh?*”或“*Aw cmon, take a chance!*”或“*You're up big, Now's the time to cash in your chips!*”。
- 3.56 编写一个 C++ 程序,利用内联函数 `circleArea`,提示用户输入圆的半径,并计算和打印圆面积。
- 3.57 说说一元作用域分辨符的用途。
- 3.58 编写一个程序,用名为 `min` 的函数模板来判断两个参数中的最小值。用整数、字符和浮点数对来测试该程序。
- 3.59 编写一个程序,用名为 `max` 的函数模板来判断两个参数中的最大值。用整数、字符和浮点数对来测试该程序。
- 3.60 判断下列程序段是否有错,并指出如何改正。注意:有的程序段中可能没有错误。

```

a) template <class A>
    int sum (int num1, int num2, int num3)
    {
        return num1 + num2 + num3;
    }

```



```
    }  
b) void printResults (int x, int y)  
    {  
        cout << "The sum is" << x+y << '\n';  
        return x+y;  
    }  
c) template <A>  
    A product (A num1, A num2, A num3)  
    {  
        return num1 * num2 * num3;  
    }  
d) double cube(int);  
    int cube (int);
```

# 第4章 数 组

## 学习目标

- 理解数组数据结构
- 理解如何用数组来保存、排序和搜索值列表和值表格
- 理解如何声明数组,如何初始化数组,以及如何对数组中的单个元素进行引用
- 学会将数组传给函数
- 理解基本的排序技术
- 会声明和操纵多下标数组

## 4.1 简介

本章对数据结构这一重要主题进行了入门性讲解。数组是一种特殊的数据结构,由一系列相同类型的相关数据项构成。在第6章,我们还会讨论结构与类的记号方法——它们各自也能容纳相关的数据项,但这些数据项可以是不同的类型。数组和结构是“静态”实体,程序执行期间,它们会保持长度不变(当然,它们也可以是自动存储类,所以当进入或离开定义它们的那些代码块时,它们就会相继地创建或删除)。在第15章,我们还会介绍能在程序执行期间增大和收缩的动态数据结构,比如列表、队列、堆栈和树等等。本章使用的数组样式是C风格的、以指针为基础的数组(将在第5章学习指针)。第8章和第20章,我们还会将数组作为一种成熟对象进行讨论,并使用面向对象的编程技术。届时,你会知道这些基于对象的数组比本章讨论的C风格的、基于指针的数组更安全、功能也更齐全。

## 4.2 数组

数组是一组连续的内存位置,它们都有相同的名称和相同的类型。为引用数组中的特定位置或元素,我们需要指定数组的名称,并指定数组中特定元素的位置编号。

图4.1显示了一个名为c的整型数组,该数组包含了12个元素。为了引用其中的一个元素,需要先给出数组的名称,后面用一对方括号([ ])将特定元素的位置编号括住。在任何数组中,第一个元素都是零元素,所以,数组c的第一个元素引用为c[0],数组c的第二个元素引用为c[1](距数组起始处1个元素),数组c的第7个元素引用为c[6](距数组起始处6个元素)。通常,数组c的第i个元素引用为c[i-1]。数组名遵守其他变量名相同的约定。

方括号内包含的位置编号有一更正式的名称,即下标。这个数字指定了距离数组起始处的元素个数,下标必须为整数,或为一个整型表达式(任何整型均可)。假如一个程序用表

数组名 (注意该数组的所有元素都有相同的名称: c)

|       |       |
|-------|-------|
| c[0]  | -45   |
| c[1]  | 6     |
| c[2]  | 0     |
| c[3]  | 72    |
| c[4]  | 1 543 |
| c[5]  | -89   |
| c[6]  | 0     |
| c[7]  | 62    |
| c[8]  | -3    |
| c[9]  | 1     |
| c[10] | 6 453 |
| c[11] | 78    |

数组c内的元素的位置编号

图 4.1 含 12 个元素的一个数组

达式作为下标,那么表达式会进行求值,判断出最终的下标是多少。例如,假定变量 a 等于 5,变量 b 等于 6,那么语句

```
c[ a + b ] += 2;
```

会让数组元素 c[11] 加 2。注意带下标的数组名是一个左值——可用于赋值语句左侧。

让我们更深入地探讨一下图 4.1 显示的数组 c。整个数组的名称是 c,它的 12 个元素分别命名为 c[0],c[1],c[2],...,c[11]。c[0] 的值是 -45,c[1] 的值是 6,c[2] 的值是 0,c[7] 的值是 62,而 c[11] 的值是 78。求和并打印数组 c 前 3 个元素的值,可使用语句

```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

为了让数组 c 第 7 个元素的值除以 2,并将结果赋给变量 x,可使用语句

```
x = c[ 6 ] / 2;
```

**常见编程错误 4.1** 有必要注意“数组第 7 个元素”和“数组元素 7”的差异。由于数组下标自 0 开始,所以“数组第 7 个元素”的下标是 6。相反,“数组元素 7”的下标就是 7,而且事实上是数组的第 8 个元素。令人遗憾的是,混淆这两种说法,往往会造成“相差 1”错误。

用于封闭数组下标的方括号实际是 C++ 的一种操作符,具有与圆括号相同的优先级。图 4.2 总结了迄今为止学过的所有操作符的优先级和结合性。优先级从上到下按降序排列,同时列出了结合性和类型。

| 操作符                        | 结合性  | 类型     |
|----------------------------|------|--------|
| ()[]                       | 从左到右 | 最高     |
| ++, --, static_cast <类型>() | 从左到右 | 一元(后缀) |
| ++, --, +, -, !            | 从右到左 | 一元(前缀) |
| *, /, %                    | 从左到右 | 乘法     |
| +, -                       | 从左到右 | 加法     |
| <<, >>                     | 从左到右 | 插入/提取  |
| <, <=, >, >=               | 从左到右 | 关系     |
| =, !=                      | 从左到右 | 相等性    |
| &&                         | 从左到右 | 逻辑 AND |
|                            | 从左到右 | 逻辑 OR  |
| ?:                         | 从右到左 | 条件     |
| =, +=, -=, *=, /=, %=      | 从右到左 | 赋值     |
| ,                          | 从左到右 | 逗号     |

图 4.2 操作符优先级和结合性

### 4.3 声明数组

数组要在内存中占据空间。程序员指定每个元素的类型以及每个数组所需的元素数量,使编译器能为其保留恰当容量的内存。为了指示编译器为整型数组 c 保留 12 个元素,要使用声明

```
int c[ 12 ];
```

可在一次声明中,同时为几个数组保留内存空间。声明

```
int b[ 100 ], x[ 27 ];
```

将为整型数组 b 保留 100 个元素,并为整型数组 x 保留 27 个元素。声明数组时,可令其包含其他数据类型。例如,类型为 char 的一个数组可用于保存一个字符串。在第 5 章,将讨论字符串及其与数组的相似性(此为 C++ 从 C 继承的一种关系),并讨论指针与数组的关系。介绍了面向对象的编程后,还会将字符串当作成熟对象进行讨论。

### 4.4 数组用法示例

图 4.3 中的程序用一个 for 重复结构初始化数组 n,把它的 10 个元素全部初始化为 0,并用表格形式打印出来。第一个输出语句显示的是列标题,为后面另一个 for 结构打印的列内容做好准备。记住 setw 指定的是下一个值即将输出的域宽。

```
1 //Fig. 4.3: fig04_03.cpp
2 //Initializing an array
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
```

```
8 #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
14     int i, n[ 10 ];
15
16     for (i=0; i<10; i++)          //initialize array
17         n[i]=0;
18
19     cout << "Element" << setw( 13 ) << "Value" << endl;
20
21     for (i = 0; i < 10; i++)
22         cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
23
24     return 0;
25 }
```

输出结果:

| Element | Value |
|---------|-------|
| 0       | 0     |
| 1       | 0     |
| 2       | 0     |
| 3       | 0     |
| 4       | 0     |
| 5       | 0     |
| 6       | 0     |
| 7       | 0     |
| 8       | 0     |
| 9       | 0     |

图 4.3 将数组元素初始化为 0

也可在数组声明中初始化数组元素,做法是在声明后添加一个等号和一个逗号分隔列表(封闭在花括号中)。注意,这个列表中的每个元素都叫做一个初始化值,整个列表统称为初始化值列表(initializer list)。图 4.4 中的程序将初始化一个含有 10 个数值的整型数组,并用表格形式打印它。

```
1 //Fig. 4.4: fig04_04.cpp
2 //Initializing an array with a declaration
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
```

```

14   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
15
16   cout << "Element" << setw( 13 ) << "Value" << endl;
17
18   for ( int i = 0; i < 10; i ++ )
19       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21   return 0;
22 }

```

输出结果:

| Element | Value |
|---------|-------|
| 0       | 32    |
| 1       | 27    |
| 2       | 64    |
| 3       | 18    |
| 4       | 95    |
| 5       | 14    |
| 6       | 90    |
| 7       | 70    |
| 8       | 60    |
| 9       | 37    |

图 4.4 在声明中初始化数组元素

假如初始化值的数量少于数组元素的数量,剩余元素会自动初始化为 0。例如,可用声明

```
int n[ 10 ] = { 0 };
```

将图 4.3 的数组 `n` 的元素初始化为 0。它显式(明确)地将第一个元素初始化为 0,并隐式(默认)地将其余元素初始化为 0,因为初始化值的个数少于数组中的元素个数。记住,自动数组不会隐式地初始化为 0。程序员至少要将第一个元素初始化为 0,剩余的其他元素才会自动变成 0。图 4.3 使用的方法可在程序执行期间反复地执行。

数组声明

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

会造成语法错误,因为一共有 6 个初始化值,事实上却只有 5 个数组元素。

**常见编程错误 4.2** 本应初始化数组元素却忘了进行初始化,是逻辑错误。

**常见编程错误 4.3** 在数组初始化列表中提供的初始化值数量多于数组元素的数量,是语法错误。

在带有一个初始化列表的数组声明中,如果省略数组长度定义,那么数组元素数量就是初始化列表的元素数量。比如声明

```
int n[] = { 1, 2, 3, 4, 5 };
```

将自动创建一个含有 5 个元素的数组。

**性能提示 4.1** 假如不是用执行时的赋值语句来初始化数组,而是在编译时用一个数组初始化列表来初始化数组,程序执行速度会更快。

图 4.5 的程序会将含有 10 个元素的数组 `s` 初始化为整数 2,4,6,...,20,并以表格形式打印这个数组。注意这些数字是每次都将循环计数器的值乘以 2,再加上 2 而获得的。

```

1 //Fig.4.5; fig04_05.cpp
2 //Initialize array s to the even integers from 2 to 20.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
14     const int arraySize = 10;
15     int j, s[ arraySize ];
16
17     for ( j = 0; j < arraySize; j ++ )    //set the values
18         s[ j ] = 2 + 2 * j;
19
20     cout << "Element" << setw( 13 ) << "Value" << endl;
21
22     for ( j = 0; j < arraySize; j ++ )    //print the values
23         cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
24
25     return 0;
26 }
```

输出结果:

| Element | Value |
|---------|-------|
| 0       | 2     |
| 1       | 4     |
| 2       | 6     |
| 3       | 8     |
| 4       | 10    |
| 5       | 12    |
| 6       | 14    |
| 7       | 16    |
| 8       | 18    |
| 9       | 20    |

图 4.5 动态生成值,并将其赋给数组元素

注意,第 14 行

```
const int arraySize = 10;
```

采用 `const` 限定符声明一个所谓的常量变量 `arraySize`,为其赋值 10。常量变量必须在声明时用 一个常量表达式进行初始化,之后不可修改(参见图 4.6 和图 4.7)。常量变量也叫做命名常量或只读变量。注意,“常量变量”显然有些自相矛盾,这在语文中称作“矛盾修饰

法”——类似于“龙虾”或“炒冰”等等。

```

1 //Fig.4.6: fig04_06.cpp
2 //Using a properly initialized constant variable
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     const int x = 7; //initialized constant variable
11
12     cout << "The value of constant variable x is: "
13         << x << endl;
14
15     return 0;
16 }
```

输出结果:

The value of constant variable x is: 7

图 4.6 常量变量的正确初始化和使用示例

```

1 //Fig.4.7: fig04_07.cpp
2 //A const object must be initialized
3
4 int main()
5 {
6     const int x; //Error: x must be initialized
7
8     x = 7;       //Error: cannot modify a const variable
9
10     return 0;
11 }
```

Borland C++ 命令行编译器输出的错误消息:

```

Fig04_07.cpp:
Error E2304 Fig04_07.cpp 6; Constant variable 'x' must be
    initialized in function main()
Error E2024 Fig04_07.cpp 8; Cannot modify a const object in
    function main()
*** 2 errors in Compile ***
```

Microsoft Visual C++ 编译器输出的错误消息:

```

Compiling...
Fig04_07.cpp
d:\fig04_07.cpp(6); error C2734:
    'x': const object must be initialized if not extern
d:\fig04_07.cpp(8); error C2166:
    1 -value specifies const object
```



Error executing cl.exe.

test.exe -2 error(s), 0 warning(s)

图 4.7 常量对象必须初始化,否则会出错

**常见编程错误 4.4** 在可执行语句中向常量变量赋值,属于语法错误。

在本来希望一个常量表达式的任何地方,都可放置常量变量。在图 4.5 中,常量变量 `arraySize` 用于声明

```
int j, s[ arraySize ];
```

中指定数组 `s` 的长度。

**常见编程错误 4.5** 只有常量才可用于声明自动和静态数组。不用常量会造成语法错误。

如用常量变量指定数组长度,可使程序具有更好的伸缩性或扩展性。在图 4.5 中,只需在 `arraySize` 的声明中,把它的值从 10 变成 1 000,第一个 `for` 循环就可以填充一个含有 1 000 个元素的数组。相反,假如没有使用常量变量 `arraySize`,那么必须在程序中的 3 个不同位置进行修改,才能对我们的程序进行扩展,使其能够处理 1 000 个数组元素。随着程序变得越来越大,为保证其条理清晰,这项技术的使用就显得越来越重要。

**软件工程知识 4.1** 将每个数组的长度定义成常量变量,但非常量可改善程序的伸缩性。

**良好编程习惯 4.1** 将一个数组的长度定义成常量变量,而不是定义成取字面值的常量,可使程序显得更清晰。利用这一技术,可有效地防范所谓的“魔数”;例如,针对一个包含 10 个元素的数组,假如在数组处理代码中,反复地提到长度 10,那么“10”这个数字就显得十分重要。一旦在程序中使用其他“10”,但它又和数组长度毫不相关,就会使读者混淆。

图 4.8 中的程序对包含 12 个元素的整型数组 `a` 的所有元素值进行求和。`for` 循环体中的语句负责实际的求和运算。有必要记住的一个重点是,作为初始化值提供给数组 `a` 的值通常是由用户通过键盘提供给程序的。比如 `for` 循环体

```
for ( int j = 0; j < arraySize; j ++ )
    cin >> a[ j ];
```

每次都从键盘读入一个值,并将其保存在元素 `a[ j ]` 中。

```
1 //Fig. 4.8: fig04_08.cpp
2 //Compute the sum of the elements of the array
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     const int arraySize = 12;
11     int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
12                          16, 45, 67, 89, 45 };
13     int total = 0;
14 }
```

```

15     for ( int i = 0; i < arraySize; i ++ )
16         total += a[ i ];
17
18     cout << "Total of array element values is " << total << endl;
19     return 0;
20 }

```

输出结果:

Total of array element values is 383

图 4.8 对数组的元素进行求和

下一个例子,将利用数组汇总问卷调查中收集的数据。问题陈述如下:

要求 40 名学生为学生食堂的饭菜质量打分,得分范围在 1~10 之间(1 表示极其糟糕,10 表示非常不错)。将 40 个得分置入一个整型数组,并对结果进行汇总。

这是一个典型的数组应用程序(参见图 4.9)。我们想汇总每种类型(这里是从 1 到 10)的回答数量。responses 是一个含有 40 个元素的数组,其中包含了所有学生的评分。我们用一个包含 11 个元素的数组 frequency 统计各种评分的产生次数。第一个元素 frequency[ 0 ] 被有意忽略,因为 1 分对应于 frequency[ 1 ] 显得比对应于 frequency[ 0 ] 更有逻辑性。这样一来,直接将各种评分当作 frequency 数组的下标即可。

```

1 //Fig. 4.9; fig04_09.cpp
2 //Student poll program
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
14     const int responseSize = 40, frequencySize = 11;
15     int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
16         10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
17         5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
18     int frequency[ frequencySize ] = { 0 };
19
20     for ( int answer = 0; answer < responseSize; answer ++ )
21         ++frequency[ responses[ answer ] ];
22
23     cout << "Rating" << setw( 17 ) << "Frequency" << endl;
24
25     for ( int rating = 1; rating < frequencySize; rating ++ )
26         cout << setw( 6 ) << rating
27             << setw( 17 ) << frequency[ rating ] << endl;
28
29     return 0;
30 }

```

输出结果:

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

图 4.9 学生投票分析程序

**良好编程习惯 4.2** 坚决保证程序的条理清晰! 有时甚至宁愿不要最有效地利用内存或处理器时间,也要编写出更清晰的程序。

**性能提示 4.2** 有时,对性能的考虑要优先于对程序清晰性的考虑。

第一个 for 循环(第 20 行和第 21 行)每次从数组 `responses` 取得一个回答(评分结果),并使 `frequency` 数组中的 10 个计数器(`frequency[ 1 ]`到 `frequency[ 10 ]`)之一自增 1。循环中最关键的语句

```
++frequency[ responses[ answer ] ];
```

令恰当的 `frequency` 计数器自增 1,具体哪一个,则取决于 `responses[ answer ]` 的值。例如,当计数器 `answer` 等于 0 时, `responses[ answer ]` 等于 1,所以 `++frequency[ responses[ answer ] ]`;实际解释为

```
++frequency[ 1 ];
```

它会使数组元素 1 自增 1。当 `answer` 等于 1 时, `responses[ answer ]` 等于 2,所以 `++frequency[ responses[ answer ] ]`;实际解释为

```
++frequency[ 2 ];
```

它会使数组元素 2 自增 1。当 `answer` 等于 2 时, `responses[ answer ]` 等于 6,所以 `++frequency[ responses[ answer ] ]`;实际解释为

```
++frequency[ 6 ];
```

它会使数组元素 6 自增 1,以此类推。注意,无论调查问卷中处理的评分数量有多少个,都只需要一个总共包含 11 个元素的数组(忽略元素 0),即可对结果进行汇总。假如数据包含像 13 这样的无效值,那么程序会试图为 `frequency[ 13 ]` 加 1。这样便超出了数组边界。注意, C++ 没有提供数组边界检查机制来防止计算机引用不存在的元素。所以,一个正在执行的程序完全有可能在不发出任何警告的前提下,超出数组的某一端,因而程序员应确保所有数组引用都保持在数组的边界之内。C++ 是一种可扩展的语言。在第 8 章,我们将利用一个类,将数组实现成一个用户自定义类型,从而对 C++ 进行扩展。有了新的数组定义后,我们可执行许多对于 C++ 内建数组来说并非标准的操作。例如,我们届时能直接比较数组,把一个数组赋给另一个,用 `cin` 和 `cout` 输入/输出整个数组,自动初始化数组,禁止对界外数组元素的访问,并能更改下标范围(甚至能更改下标类型),使数组的第一个元素不必为 0。

**常见编程错误 4.6** 引用数组边界之外的元素,会造成执行时逻辑错误。但这不属于语法错误。

**测试和调试提示 4.1** 遍历一个数组时,数组下标永远不能小于0,而且永远都要小于数组中的元素总数(比数组长度小1)。请在循环中止条件中,杜绝访问超出这个范围的元素。

**测试和调试提示 4.2** 程序应检查所有输入值的正确性,防止错误信息影响程序计算。

**可移植性提示 4.1** 如引用数组边界之外的元素,所造成的影响(通常都是严重的)与具体系统有关,这通常会修改一个无关变量的值。

**测试和调试提示 4.3** 等我们学习类时(从第6章开始),将知道如何开发一个“智能数组”,它能在运行时自动检查所有下标引用都在边界之内。使用像这样的智能数据类型,有助于消除程序中的 Bug。

我们的下一个例子(图 4.10)将从一个数组读入数字,并以条形图(或直方图)的形式,对信息进行图示——每个数字都会打印下来,然后在数字旁边,打印一个直方,其中恰好包含了那个数量的星号。嵌套的 for 循环将负责实际描绘直方图。注意最后要用一个 endl 中止描绘直方图。

```

1 //Fig. 4.10: fig04_10.cpp
2 //Histogram printing program
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
14     const int arraySize = 10;
15     int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
16
17     cout << "Element" << setw( 13 ) << "Value"
18         << setw( 17 ) << "Histogram" << endl;
19
20     for ( int i = 0; i < arraySize; i ++ ) {
21         cout << setw( 7 ) << i << setw( 13 )
22             << n[ i ] << setw( 9 );
23
24         for ( int j = 0; j < n[ i ]; j ++ ) //print one bar
25             cout << '*';
26
27         cout << endl;
28     }
29

```

```

30     return 0;
31 }

```

输出结果:

| Element | Value | Histogram |
|---------|-------|-----------|
| 0       | 13    | *****     |
| 1       | 3     | ***       |
| 2       | 15    | *****     |
| 3       | 7     | *****     |
| 4       | 11    | *****     |
| 5       | 9     | *****     |
| 6       | 13    | *****     |
| 7       | 5     | *****     |
| 8       | 17    | *****     |
| 9       | 1     | *         |

图 4.10 打印直方图的一个程序

**常见编程错误 4.7** 尽管可能在一个 for 循环和另一个嵌套于其中的 for 循环中使用相同的计数器变量,但这样常会造成逻辑错误。

**测试和调试提示 4.4** 尽管可在一个 for 主体中修改一个循环计数器,但尽量避免这样做,因为这通常都会导致容易被人忽视的 Bug。

第 3 章曾提过,我们会展示一个更优雅的方法来编写图 3.8 的掷骰子程序。现在的问题是,要丢掷 6 000 次一枚六面骰子,以检测随机数生成器是否真的产生随机数。图 4.11 展示了这个程序的数组版本。

```

1 //Fig. 4.11: fig04_11.cpp
2 //Roll a six-sided die 6000 times
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>
13 #include <ctime>
14
15 int main()
16 {
17     const int arraySize = 7;
18     int face, frequency[ arraySize ] = { 0 };
19
20     srand( time( 0 ) );
21
22     for ( int roll = 1; roll <= 6000; roll ++ )
23         ++frequency[ 1 + rand() % 6 ]; //replaces 20-line switch

```

```

24                                     //of Fig. 3.8
25
26     cout << "Face" << setw( 13 ) << "Frequency" << endl;
27
28     //ignore element 0 in the frequency array
29     for ( face = 1; face < arraySize; face ++ )
30         cout << setw( 4 ) << face
31             << setw( 13 ) << frequency[ face ] << endl;
32
33     return 0;
34 |

```

输出结果:

| Face | Frequency |
|------|-----------|
| 1    | 1037      |
| 2    | 987       |
| 3    | 1013      |
| 4    | 1028      |
| 5    | 952       |
| 6    | 983       |

图 4.11 用数组而不是 switch 来掷骰子

到目前为止,我们讨论的还只限于整型数组。不过,数据实际可为任意类型。现在,我们打算讨论将字符串保存到字符数组的情况。迄今为止,我们讲解过的惟一字符串处理功能便是用 cout 和 << 来输出一个字符串。对于像“hello”这样的一个字符串来说,它实际就是一个字符数组。字符数组具有几个独一无二的特性。

字符数组可用一个字符字面值来初始化。比如声明

```
char string1[] = "first";
```

会将数组 string1 的元素初始化为“first”这个字符串字面值中的各个字符。在上述声明中, string1 数组的长度是由编译器根据字符串长度决定的。要注意的一个重点在于,在字符串“first”中,除包含 5 个字符之外,还要加上一个特殊的字符串结束字符,即 NULL 字符。所以, string1 这个数组实际包含了 6 个元素。用于表示 NULL 字符的字符常量是 '\0' (反斜杠后跟一个 0),所有字符串都是用这个字符结束的。用于表示一个字符串的字符数组应当声明得足够大,保证容下字符串中的所有字符再加上一个 NULL 结束字符。

字符数组还可用一个初始化列表中的单个字符常量进行初始化。前述的声明等价于下面这种乏味得多的形式

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

由于字符串就是一个字符数组,所以可使用数组下标记号法,直接访问一个字符串中的单个字符。举个例子来说, string1[0] 就是字符 'f', 而 string1[3] 就是字符 's'。

还可利用 cin 和 >>, 将一个字符串从键盘直接输入字符数组中。比如声明

```
char string2[ 20 ];
```

创建了一个字符数组,能够存 19 个普通字符,再加 1 个 NULL 结束字符。语句

```
cin >> string2;
```

将从键盘把一个字符串读入 string2,并在 string2 的末尾自动追加 NULL 字符。注意在上述语句中,只提供了数组名,没有提供同数组长度有关的任何信息。在此,要由程序员负责保

证数组能容下用户通过键盘输入的任何字符串。cin 会一直从键盘读取字符,直到遇到第一个空白字符为止——它并不关心数组到底有多大。所以,用 cin 和 >> 输入数据时,有可能造成数据超出数组的边界(参见 5.12 节,了解如何防止插入的内容越过 char 数组的末端)。

**常见编程错误 4.8** 用 cin >> 输入数据时,若键盘向字符数组输入超出其容量的字符串,会造成程序数据丢失,以及其他严重的运行错误。

可用 cout 和 << 输出一个字符数组,用它表示一个以 NULL 结束的字符串。string2 这个数组可用语句

```
cout << string2 << endl;
```

打印。注意,类似 cin >> 和 cout << 并不关心字符数组到底有多大,字符串中的字符会连续打印,直到碰到 NULL 结束字符为止。

图 4.12 展示了用字符串字面值对字符数组进行初始化的情况,把字符串读入字符数组,把字符数组作为一个字符串打印,最后访问字符串中的单独字符。

```
1 //Fig. 4_12; fig04_12.cpp
2 //Treating character arrays as strings
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     char string1[ 20 ], string2[] = "string literal";
12
13     cout << "Enter a string: ";
14     cin >> string1;
15     cout << "string1 is: " << string1
16         << " \nstring2 is: " << string2
17         << " \nstring1 with spaces between characters is: \n";
18
19     for ( int i = 0; string1[ i ] != '\0'; i ++ )
20         cout << string1[ i ] << " ";
21
22     cin >> string1; //reads "there"
23     cout << " \nstring1 is: " << string1 << endl;
24
25     cout << endl;
26     return 0;
27 }
```

输出结果:

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
```

```

Hello
string1 is: there

```

图 4.12 将字符数组当作字符串处理

图 4.12 用一个 for 结构(第 19 行和第 20 行)遍历 string1 数组,并打印其中每个字符,中间用一个空格分隔。对于 for 结构中的条件来说——string1[ i ] != '\0'——只要没在字符串中遇到 NULL 结束符,它就会保持为 True。

第 3 章讨论了存储类说明符 static。函数定义中的静态局部变量会在程序运行期间一直存在,但只能用于函数体内。

**性能提示 4.3** 可向一个局部数组声明应用 static,使数组不至于在每次调用函数时都进行创建和初始化。同时,当每次函数在程序中退出时,也不至于被删除。这样可提高性能。

声明为 static 的数组是在程序载入时初始化的。假如 static 数组不是由程序员显式地初始化,那么在创建那个数组时,编译器会自动将其初始化为零。

图 4.13 展示了函数 staticArrayInit 带有一个声明为 static 的局部数组,函数 automaticArrayInit 带有一个自动局部数组,函数 staticArrayInit 被调用了两次。编译器将 static 局部数组初始化为零。函数打印数组,为其中每个元素加 5,并再次打印数组。第二次调用函数时,静态数组包含了在第一次函数调用时保存下来的值。函数 automaticArrayInit 也被调用了两次。自动局部数组的元素是用 1,2 和 3 这几个值初始化的。函数打印数组,为每个元素加 5,并再次打印数组。第二次调用函数时,数组元素会被重新初始化为 1,2 和 3,因为数组使用的是自动存储类。

```

1 //Fig. 4.13: fig04_13.cpp
2 //Static arrays are initialized to zero
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void staticArrayInit( void );
9 void automaticArrayInit( void );
10
11 int main()
12 |
13     cout << "First call to each function:\n";
14     staticArrayInit();
15     automaticArrayInit();
16
17     cout << "\n\nSecond call to each function:\n";
18     staticArrayInit();
19     automaticArrayInit();
20     cout << endl;
21
22     return 0;
23 |
24

```



```

25 //function to demonstrate a static local array
26 void staticArrayInit( void )
27 {
28     static int array1[ 3 ];
29     int i;
30
31     cout << "\nValues on entering staticArrayInit:\n";
32
33     for ( i = 0; i < 3; i ++ )
34         cout << "array1[" << i << "] = " << array1[ i ] << " ";
35
36     cout << "\nValues on exiting staticArrayInit:\n";
37
38     for ( i = 0; i < 3; i ++ )
39         cout << "array1[" << i << "] = "
40             << ( array1[ i ] += 5 ) << " ";
41 }
42
43 //function to demonstrate an automatic local array
44 void automaticArrayInit( void )
45 {
46     int i, array2[ 3 ] = { 1, 2, 3 };
47
48     cout << "\n\nValues on entering automaticArrayInit:\n";
49
50     for ( i = 0; i < 3; i ++ )
51         cout << "array2[" << i << "] = " << array2[ i ] << " ";
52
53     cout << "\nValues on exiting automaticArrayInit:\n";
54
55     for ( i = 0; i < 3; i ++ )
56         cout << "array2[" << i << "] = "
57             << ( array2[ i ] += 5 ) << " ";
58 }

```

#### 输出结果:

First call to each function:

Values on entering staticArrayInit:  
array1[0]=0 array1[1]=0 array1[2]=0  
Values on exiting staticArrayInit:  
array1[0]=5 array1[1]=5 array1[2]=5

Values on entering automaticArrayInit:  
array2[0]=1 array2[1]=2 array2[2]=3  
Values on exiting automaticArrayInit:  
array2[0]=5 array2[1]=7 array2[2]=8

Second call to each function:

Values on entering staticArrayInit:  
array1[0]=5 array1[1]=5 array1[2]=5  
Values on exiting staticArrayInit:  
array1[0]=10 array1[1]=10 array1[2]=10

```

Values on entering automaticArrayInit:
array2[0] =1  array2[1] =2  array2[2] =3
Values on exiting automaticArrayInit:
array2[0] =6  array2[1] =7  array2[2] =8

```

图 4.13 static 数组初始化与自动数组初始化的对比

**常见编程错误 4.9** 假定每次调用一个函数时,它的局部 static 数组的元素都重新初始化为零,有可能导致程序中产生逻辑错误。

## 4.5 将数组传给函数

为了将一个数组实参传给函数,需要在不使用方括号的前提下指定数组名。例如,假定数组 `hourlyTemperatures` 的声明如下

```
int hourlyTemperatures[ 24 ];
```

函数调用语句

```
modifyArray( hourlyTemperatures, 24 );
```

会将数组 `hourlyTemperatures` 及其长度传给函数 `modifyArray`。向函数传递一个数组时,数组长度通常也要传递,目的是使函数能处理数组中特定数量的元素(否则,我们需要在被调用的函数中建立这种信息;或者更糟,需要把数组长度放在一个全局变量中)。在第 8 章,当我们介绍 `Array` 类时,会将数组长度构建到用户自定义类型中——这样一来,我们创建的每个 `Array` 对象都会“知道”自己的长度。届时再将 `Array` 对象传给函数时,便不必将数组长度作为实参传递了。

C++ 会使用模拟的“按引用调用”(call-by-reference)将数组传给函数——被调用的函数能修改调用者的原始数组的元素值。数组名的值就是数组第一个元素的地址。由于数组的起始地址已经传递,所以被调用的函数准确地知道数组存放于何处。所以,当被调用的函数在函数体中修改数组元素时,就相当于直接修改位于原始内存位置的实际数组元素。

**性能提示 4.4** 通过模拟的“按引用调用”来传递数组,可在一定程度上改善性能。假如数组通过“传值调用”(call-by-value)进行传递,那么需要传递每个元素的一个副本。对于大型的、经常传递的数组来说,这显然既会浪费时间,又会浪费可观的存储空间来保存数组副本。

**软件工程知识 4.2** 完全有可能按值传递一个数组(利用第 16 章解释的简单技巧)——尽管极少有人会这样做。

尽管整个数组是通过模拟的“按引用调用”来传递的,但其中单独的数组元素却是通过“按值调用”来传递的,这和简单变量是一样的,像这样的简单数据元素叫做标量(scalars 或 scalar quantities)。为了向函数传递一个数组元素,要在函数调用中使用数组元素的下标名称作为一个实参。在第 5 章,我们会展示如何为标量(亦即单独的变量和数组元素)模拟“按引用调用”。

函数为了通过一个函数调用来接收一个数组,在函数的参数列表中,必须指出打算接收一个数组。例如,函数 `modifyArray` 的函数头可写为

```
void modifyArray( int b[], int arraySize )
```

它表明, `modifyArray` 希望在参数 `b` 中, 接收一个整型数组的地址; 并在参数 `arraySize` 中, 接收数组元素的数量。在数组方括号内, 是不需要添加数组长度的。即使包括了它, 编译器也会忽略。由于数组是通过模拟的“按引用调用”来传递的, 所以一旦被调用的函数使用数组名 `b`, 它事实上引用的是调用者中的实际数组(上述调用中的数组 `hourlyTemperatures`)。在第 5 章, 我们还会讲解如何利用其他记号法来表示函数接收一个数组的情况。如同届时会看到的那样, 这些记号法均是以数组和指针间的密切关系为基础的。

请注意 `modifyArray` 的函数原型的奇怪写法

```
void modifyArray( int [], int );
```

这个原型也可写为

```
void modifyArray( int anyArrayName[], int anyVariableName);
```

但正如我们在第 3 章学到的那样, C++ 编译器会忽略原型中的变量名。

**良好编程习惯 4.3** 有的程序员在函数原型中包括变量名, 目的是使程序更清晰。注意编译器会忽略这些名称。

记住, 原型的作用只是告诉编译器实参的数量, 以及每个实参的类型(按实参预期的出现顺序排列)。

图 4.14 的程序展示了传递整个数组同传递一个数组元素的差异。程序首先打印整型数组 `a` 的 5 个元素。接着, `a` 及其长度会传给函数 `modifyArray`。在函数中, `a` 的每个元素都会乘以 2。然后, 在 `main` 中重新打印 `a`。如输出结果所示, `a` 的元素真的被 `modifyArray` 修改了。现在, 程序会打印 `a[3]` 的值, 并将其传给函数 `modifyElement`。这个函数会让它的实参乘以 2, 然后打印新值。注意, 当 `a[3]` 在 `main` 中重新打印时, 它尚未被修改, 因为单独的数组元素是“按值调用”来传递的。

```
1 //Fig. 4.14: fig04_14.cpp
2 //Passing arrays and individual array elements to functions
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 void modifyArray( int [], int ); //appears strange
13 void modifyElement( int );
14
15 int main()
16 {
17     const int arraySize = 5;
18     int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
19
20     cout << "Effects of passing entire array call-by-reference: "
21          << " \n\nThe values of the original array are: \n";
22
23     for ( i = 0; i < arraySize; i ++ )
```

```

24     cout << setw( 3 ) << a[ i ];
25
26     cout << endl;
27
28     //array a passed call-by-reference
29     modifyArray( a, arraySize );
30
31     cout << "The values of the modified array are;\n";
32
33     for ( i = 0; i < arraySize; i ++ )
34         cout << setw( 3 ) << a[ i ];
35
36     cout << "\n\n\n"
37         << "Effects of passing array element call-by-value: "
38         << "\nThe value of a[3] is " << a[ 3 ] << '\n';
39
40     modifyElement( a[ 3 ] );
41
42     cout << "The value of a[3] is " << a[ 3 ] << endl;
43
44     return 0;
45 }
46
47 //In function modifyArray, "b" points to the original
48 //array "a" in memory.
49 void modifyArray( int b[], int sizeofArray )
50 {
51     for ( int j = 0; j < sizeofArray; j ++ )
52         b[ j ] *= 2;
53 }
54
55 //In function modifyElement, "e" is a local copy of
56 //array element a[ 3 ] passed from main.
57 void modifyElement( int e )
58 {
59     cout << "Value in modifyElement is "
60         << ( e *= 2 ) << endl;
61 }

```

输出结果:

Effects of passing entire array call-by-reference;

The values of the original array are;

0 1 2 3 4

The values of the modified array are;

0 2 4 6 8

Effects of passing array element call-by-value;

The value of a [3] is 6

Value in modifyElement is 12

The value of a[3] is 6

图 4.14 向数组传递数组和单独的数组元素

在你的程序中,可能会出现遇到不允许函数修改数组元素的情况。由于数组肯定是通过模拟的“按引用调用”而传递的,所以对数组中的值进行修改时较难控制。C++ 提供了类型限定符 `const`, 可用于防止在函数中对数组进行修改。假如一个函数在指定一个数组参数的同时,在它前面加了一个 `const` 限定符,那么数组元素会在函数体内成为常量。以后,任何在函数体中对数组元素进行修改的企图都会导致语法错误。根据错误提示,程序员就可改正程序,使其不要试图修改数组元素。

图 4.15 演示了 `const` 限定符的使用情况。在定义函数 `tryToModifyArray` 时,指定了参数 `const int b[]`, 它指出数组 `b` 为常量,不可修改。函数对数组元素的 3 次尝试都会出现语法错误:“Cannot modify a const object”(不可修改一个常量对象)。`const` 限定符将在第 7 章继续讨论。

```

1 //Fig. 4.15: fig04_15.cpp
2 //Demonstrating the const type qualifier
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void tryToModifyArray( const int [] );
9
10 int main()
11 {
12     int a[] = { 10, 20, 30 };
13
14     tryToModifyArray( a );
15     cout << a[ 0 ] << " " << a[ 1 ] << " " << a[ 2 ] << '\n';
16     return 0;
17 }
18
19 //In function tryToModifyArray, "b" cannot be used
20 //to modify the original array "a" in main.
21 void tryToModifyArray( const int b[] )
22 {
23     b[ 0 ] /= 2;    //error
24     b[ 1 ] /= 2;    //error
25     b[ 2 ] /= 2;    //error
26 }
```

Borland C++ 命令行编译器输出的错误消息:

```

Fig04_15.cpp:
Error E2024 Fig04_15.cpp 23: Cannot modify a const object in
    function tryToModifyArray(const int * const)
Error E2024 Fig04_15.cpp 24: Cannot modify a const object in
    function tryToModifyArray (const int * const)
Error E2024 Fig04_15.cpp 25: Cannot modify a const object in
    function tryToModifyArray (const int * const)
Warning W8057 Fig04_15.cpp 26: Parameter 'b' is never used in
    function tryToModifyArray (const int * const)
*** 3 errors in Compile ***
```

Microsoft Visual C++ 编译器输出的错误消息:

```
Compiling...
Fig04_15.cpp
D:\Fig04_15.cpp(23): error C2166:
  1 - value specifies const object
D:\Fig04_15.cpp(24): error C2166:
  1 - value specifies const object
D:\Fig04_15.cpp(25): error C2166:
  1 - value specifies const object
Error executing cl.exe.
test.exe -3 error(s), 0 warning(s)
```

图 4.15 演示 const 类型限定符

**常见编程错误 4.10** 如忘了数组是按引用传递的(所以是完全能够修改的),有可能造成逻辑错误。

**软件工程知识 4.3** const 类型限定符可应用于函数定义中的一个数组参数,从而防止原始数组在函数体中被不慎修改。这是“最低权限”原则的另一个例子。除非绝对需要,否则不要为赋予函数修改数组的权力。

## 4.6 数组排序

数据的排序(换言之,按特定的顺序排列数据,比如按升序或降序等等)是最重要的计算应用之一。一家银行需要按账户号码对所有支票进行排序,以便在每个月末,准备个人的银行报表。电话公司需要先按姓氏对账户列表进行排序,再按名字进行排序,以便更容易地找到电话号码。几乎每个单位都必须对一些数据进行排序(而且通常是大量数据)。对数据进行排序是一个颇为有趣的问题,计算机科学领域的许多研究都是围绕它而展开的。在本章,我们将讨论最简单的已知排序方案。在本章的练习题和第 15 章中,我们将探讨更复杂的方案,追求更出色的排序性能。

**性能提示 4.5** 有时,最简单的算法在性能上也是最差的。它们惟一的优点便是容易编写、测试和调试。为了获得最好的性能,往往需要采取更复杂的算法。

图 4.16 中的程序负责按升序对总共包含 10 个元素的数组 a 的值进行排序。在此采用的技术叫做冒泡排序(bubble sort)或者下沉排序(sinking sort),因为越来越小的值会像水中的气泡一样,逐渐“冒”到数组的顶部;同时越来越大的值会逐渐“沉”到数组的底部。采取这种排序技术,会在数组中连续经历多个步骤。每一步,都会有比较连续的元素对。假如某一对元素呈现递增顺序(或者两个值相等),则保持这两个值不变;如某一对元素呈现递减顺序,两个值会在数组中交换位置。

```
1 //Fig. 4.16: fig04_16.cpp
2 //This program sorts an array's values into
3 //ascending order
4 #include <iostream>
5
```

```

6  using std::cout;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 int main()
14 {
15     const int arraySize = 10;
16     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
17     int i, hold;
18
19     cout << "Data items in original order\n";
20
21     for ( i = 0; i < arraySize; i ++ )
22         cout << setw( 4 ) << a[ i ];
23
24     for ( int pass = 0; pass < arraySize - 1; pass ++ ) //passes
25
26         for ( i = 0; i < arraySize - 1; i ++ )          //one pass
27
28             if ( a[ i ] > a[ i + 1 ] ) {                 //one comparison
29                 hold = a[ i ];                          //one swap
30                 a[ i ] = a[ i + 1 ];
31                 a[ i + 1 ] = hold;
32             }
33
34     cout << "\nData items in ascending order\n";
35
36     for ( i = 0; i < arraySize; i ++ )
37         cout << setw( 4 ) << a[ i ];
38
39     cout << endl;
40     return 0;
41 }

```

输出结果:

```

Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in ascending order
 2  4  6  8 10 12 37 45 68 89

```

图 4.16 用“冒泡排序”法对数组进行排序

程序首先会比较  $a[0]$  和  $a[1]$ , 接着比较  $a[1]$  和  $a[2]$ , 接着比较  $a[2]$  和  $a[3]$ , ... 以此类推, 直到完成了  $a[8]$  同  $a[9]$  的比较为止。尽管总共有 10 个元素, 但却执行了 9 次比较。由于是连续比较, 所以每一次, 一个大值都可能在数组中向下移动多个位置, 而一个小值只能向上移动一个位置。第 1 次, 最大值会逐渐下沉到数组的底部元素  $a[9]$ ; 第 2 次, 第 2 个大值会逐渐下沉到  $a[8]$ 。在第 9 次, 第 9 个大值会下沉到  $a[1]$ 。最后, 最小值

便留在 `a[0]` 中。因此,只需 9 次,即可完成对一个总共 10 个元素的数组的排序。

排序是通过嵌套的 `for` 循环来执行的。假如需要进行一次位置交换,可通过 3 次赋值操作

```
hold = a[ i ];
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = hold;
```

来完成。其中,附加的变量 `hold` 用于临时存储要交换的两个值之一。如果只使用两次赋值

```
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = a[ i ];
```

交换是无法完成的。例如,假定 `a[i]` 等于 7,而 `a[i + 1]` 等于 5,那么在第一次赋值后,两个值都是 5,值 7 就丢失了,这正是为何需要附加变量 `hold` 的原因。

冒泡排序的主要优点便是易于编程。不过,冒泡排序的速度实在太慢。在对大型数组排序时,便会深刻体会到这一点。在本章的练习中,我们会开发冒泡排序的一个更高效的版本,并探讨比冒泡排序有效得多的其他一些排序方法。在更高级的课程中,还会针对排序和搜索问题展开更加深入的学习。

## 4.7 案例分析:利用数组计算均数、中位数和众数

现在来分析一个更大的例子。我们常用计算机编辑和分析调查及投票结果。图 4.17 中的程序使用了数组 `response`,它用一次问卷调查的 99 个结果(99 用常量变量 `responseSize` 表示)。每个结果都是 1~9 之间的数字之一。程序将分别计算这 99 个值的均数、中位数和众数。

```
1 //Fig. 4.17: fig04_17.cpp
2 //This program introduces the topic of survey data analysis.
3 //It computes the mean, median, and mode of the data.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <iomanip>
11
12 using std::setw;
13 using std::setiosflags;
14 using std::setprecision;
15
16 void mean( const int [], int );
17 void median( int [], int );
18 void mode( int [], int [], int );
19 void bubbleSort( int[], int );
```



```

20 void printArray( const int[], int );
21
22 int main()
23 {
24     const int responseSize = 99;
25     int frequency[ 10 ] = { 0 },
26         response[ responseSize ] =
27         { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
28           7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
29           6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
30           7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
31           6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
32           7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
33           5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
34           7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
35           7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
36           4, 5, 6, 1, 6, 5, 7, 8, 7 };
37
38     mean( response, responseSize );
39     median( response, responseSize );
40     mode( frequency, response, responseSize );
41
42     return 0;
43 }
44
45 void mean( const int answer[], int arraySize )
46 {
47     int total = 0;
48
49     cout << " * * * * * \n Mean \n * * * * * \n";
50
51     for ( int j = 0; j < arraySize; j ++ )
52         total += answer[ j ];
53
54     cout << "The mean is the average value of the data \n"
55          << "items. The mean is equal to the total of \n"
56          << "all the data items divided by the number \n"
57          << "of data items ( " << arraySize
58          << "). The mean value for \nthis run is: "
59          << total << " / " << arraySize << " = "
60          << setiosflags( ios::fixed | ios::showpoint )
61          << setprecision( 4 )
62          << static_cast< double >( total ) / arraySize << " \n \n";
63 }
64
65 void median( int answer[], int size )
66 {
67     cout << " \n * * * * * \n Median \n * * * * * \n"
68          << "The unsorted array of responses is";
69
70     printArray( answer, size );

```

```

71  bubbleSort( answer, size );
72  cout << "\n\nThe sorted array is";
73  printArray( answer, size );
74  cout << "\n\nThe median is element " << size /2
75      << " of \nthe sorted " << size
76      << " element array.\nFor this run the median is "
77      << answer[ size /2 ] << "\n\n";
78  |
79
80  void mode( int freq[], int answer[], int size )
81  |
82      int rating, largest = 0, modeValue = 0;
83
84      cout << "\n*****\n Mode\n*****\n";
85
86      for ( rating = 1; rating <= 9; rating ++ )
87          freq[ rating ] = 0;
88
89      for ( int j = 0; j < size; j ++ )
90          ++freq[ answer[ j ] ];
91
92      cout << "Response" << setw( 11 ) << "Frequency"
93          << setw( 19 ) << "Histogram\n\n" << setw( 55 )
94          << "1    1    2    2\n" << setw( 56 )
95          << "5    0    5    0    5\n\n";
96
97      for ( rating = 1; rating <= 9; rating ++ ) {
98          cout << setw( 8 ) << rating << setw( 11 )
99              << freq[ rating ] << "          ";
100
101          if ( freq[ rating ] > largest ) {
102              largest = freq[ rating ];
103              modeValue = rating;
104          }
105
106          for ( int h = 1; h <= freq[ rating ]; h ++ )
107              cout << '*';
108
109          cout << '\n';
110      }
111
112      cout << "The mode is the most frequent value.\n"
113          << "For this run the mode is " << modeValue
114          << " which occurred " << largest << " times, " << endl;
115  }
116
117  void bubbleSort( int a[], int size )
118  |
119      int hold;
120

```

```

121     for ( int pass = 1; pass < size; pass ++ )
122
123         for ( int j = 0; j < size - 1; j ++ )
124
125             if ( a[ j ] > a[ j + 1 ] ) {
126                 hold = a[ j ];
127                 a[ j ] = a[ j + 1 ];
128                 a[ j + 1 ] = hold;
129             }
130     }
131
132     void printArray( const int a[], int size )
133     {
134         for ( int j = 0; j < size; j ++ ) {
135
136             if ( j % 20 == 0 )
137                 cout << endl;
138
139             cout << setw( 2 ) << a[ j ];
140         }
141     }

```

图 4.17 调查问卷数据分析程序

所谓“均数”，是指 99 个值的算术平均值。函数 `mean` 为了计算均数，将先求 99 个元素的和，再将结果除以 99。

所谓“中位数”，是指对结果排序后，位于最中间的那个值，即“中间值”。函数 `median` 为了判断中位数，需要调用 `bubbleSort` 对数组 `response` 进行排序，并在排好序的数组中，挑选出最中间的元素——`answer[ size / 2 ]`。注意，假如元素数量是一个偶数，那么中位数应当计算成两个中间元素的均数。函数 `median` 未提供这样的功能。调用函数 `printArray`，以便输出数组 `response`。

所谓“众数”，是指 99 个结果中出现得最频繁的那一个。函数 `mode` 会统计每种类型的结果数量，然后选择计数最大的那个值。注意这个版本的 `mode` 函数并没有处理出现平局的情况（参见练习题 4.14）。函数 `mode` 还会生成一个直方图，以图形化的方式判断众数。图 4.18 包含了这个程序的一次示范运行的情况。该例包含了在解决数组问题时，会牵涉到的大多数常规操作，其中也包括向函数传递数组的情况。

输出结果：

```

*****
Mean
*****
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788

```

```

*****
    Median
*****
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 8

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 9 9 9 9 9 9 9 9 7 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
The sorted 99 element array.
For this run the median is 7

*****
    Mode
*****
Response    Frequency    Histogram

                                1    1    2    2
                                5    0    5    0    5

    1          1          *
    2          3          ***
    3          4          ****
    4          5          *****
    4          5          *****
    5          8          *********
    6          9          **********
    7          23         *********************
    8          27         *************************
    9          19         *****************

The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.

```

图 4.18 “调查问卷”结果数据分析程序的示范执行情况

## 4.8 搜索数组:线性搜索和二元搜索

程序员经常都要处理数组中保存的大量数据。可能需要判断一个数组是否包含了同特定键值对应的一个值。查找一个特定数组元素的过程叫做搜索。在本节,我们将讨论两种搜索技术——简单的线性搜索技术和更有效的二元搜索技术。本章末尾的练习题 4.33 和 4.34 会要求你实现线性搜索和二元搜索的迭代版本。

线性搜索(图 4.19)会将数组的每个元素同搜索键进行对比。由于数组元素并非按照特定的顺序排列,所以既有可能在第一个元素中便能找到与搜索键相符的值,也有可能到最后—

个元素才找到。所以,程序必须拿数组里一半的元素同指定的搜索键进行对比。运气特别差时,程序甚至要一一比较数组中的每一个元素,最后发现根本没有匹配的元素。

```

1 //Fig.4.19; fig04_19.cpp
2 //Linear search of an array
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int linearSearch( const int [], int, int );
10
11 int main()
12 {
13     const int arraySize = 100;
14     int a[ arraySize ], searchKey, element;
15
16     for ( int x = 0; x < arraySize; x ++ ) //create some data
17         a[ x ] = 2 * x;
18
19     cout << "Enter integer search key: " << endl;
20     cin >> searchKey;
21     element = linearSearch( a, searchKey, arraySize );
22
23     if ( element != -1 )
24         cout << "Found value in element " << element << endl;
25     else
26         cout << "Value not found" << endl;
27
28     return 0;
29 }
30
31 int linearSearch( const int array[], int key, int sizeOfArray )
32 {
33     for ( int n = 0; n < sizeOfArray; n ++ )
34         if ( array[ n ] == key )
35             return n;
36
37     return -1;
38 }

```

输出结果:

```

Enter integer search key:
36
Found value in element 18

Enter integer search key:
37
Value not Found

```

图 4.19 对一个数组进行线性搜索

线性搜索方法特别适用小数组或未排序的数组。但是,对于大型数组来说,线性搜索的效率未免太低。假如数组已经排序,就应选择高速的二元搜索技术。

采用二元搜索算法,每进行一次比较,都可将搜索数组内的元素排除掉一半。该算法首先会找到数组中部的元素,把它同搜索键进行比较。如相等,表明找到了搜索键,并返回那个元素的数组下标。否则,便开始搜索数组的一半元素——假如搜索键小于数组中部的元素,则搜索数组前半部分的元素,否则搜索后半部分的元素。接下来,假如搜索键不是当前子数组(乃原始数组的一个子集)的中部元素,那么类似地,该算法会再将这个子数组分成两半,并在四分之一原始数组的范围内搜索。这种操作会一直这样继续下去,直至搜索键最终等于了某个子数组的中部元素,或数组经不断拆分后,变得只剩下一个元素,而且该元素不与搜索键相等(如出现的是后一种情况,表明没有发现搜索键)。

即便在运气最差的情况下,搜索包含 1 024 个元素的数组,最多也只需 10 次比较。为什么呢? 因为 1 024 会不断地除以 2(每次搜索后,都可排除掉一半元素),所以连续生成了 512, 256, 126, 64, 32, 16, 8, 4, 2 和 1 个子数组。1 024( $2^{10}$ )只需连续被 2 除 10 次,便会得到值 1。每被 2 除一次,便相当于二元搜索算法中的一次比较。对一个总共包含了 1 048 576 ( $2^{20}$ ) 个元素的数组来说,最多 20 次比较,便能找到搜索键(或证明没有找到)。而对包含了 10 亿个元素的数组来说,最多也只需 30 次比较,便能完成搜索。显然,同前述线性搜索技术相比,二元搜索可大幅地提升效率与性能。后者要求将搜索键同数组内平均一半的元素进行对比。对包含了 10 亿元素的数组来说,5 亿次和最多 30 次对比可存在着天壤之别! 对任何排好序的数组进行二元搜索时,要想提前知道所需的最大对比次数,可试着计算 2 的  $n$  次方结果( $n$  依次增大),只要首次有一个结果大于数组内的元素数量,当时的  $n$  值便是最多需要的对比次数。

**性能提示 4.6** 二元搜索相较线性搜索所带来的巨大性能提升也不是没有代价的。对数组进行排序本身便是一个耗时耗力的工作,它比每次搜索一个项目要“费劲儿”得多。不过,假如需要不断搜索一个数组,同时又想获得最好的性能,那么提前为它排好序仍是颇为值得的。

图 4.20 展示了函数 `binarySearch` 的迭代版本。函数会接收 4 个实参——一个整型数组 `b`, 一个整数 `searchKey`, `low` 数组下标以及 `high` 数组下标。假如搜索键与子数组的中部元素不相符,就会调节 `low` 或 `high` 下标,以便对较小的子数组进行搜索。如搜索键小于中部元素,那么 `high` 下标会被设为 `middle - 1`,并会继续搜索从 `low` 到 `middle - 1` 的元素。如搜索键大于中部元素,那么 `low` 下标会被设为 `middle + 1`,并会继续搜索从 `middle + 1` 到 `high` 的元素。程序使用了总共包含 15 个元素的一个数组。大于数组元素数量的第一个 2 的  $n$  次方结果是 16,此时  $n$  等于 4( $2^4 = 16$ )。所以,最多只需 4 次比较,即可找到搜索键。函数 `printHeader` 会输出数组下标,而函数 `printRow` 会输出二元搜索过程中的每一个子数组。每个子数组的中部元素用一个星号(\*)标记,指出同搜索键比较的是一个元素。

```
1 //Fig. 4.20: fig04_20.cpp
2 //Binary search of an array
3 #include <iostream>
4
```

```
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 int binarySearch( const int [], int, int, int, int );
14 void printHeader( int );
15 void printRow( const int [], int, int, int, int );
16
17 int main()
18 |
19     const int arraySize = 15;
20     int a[ arraySize ], key, result;
21
22     for ( int i = 0; i < arraySize; i ++ )
23         a[ i ] = 2 * i;    //place some data in array
24
25     cout << "Enter a number between 0 and 28: ";
26     cin >> key;
27
28     printHeader( arraySize );
29     result = binarySearch( a, key, 0, arraySize - 1, arraySize );
30
31     if ( result != -1 )
32         cout << '\n' << key << " found in array element "
33             << result << endl;
34     else
35         cout << '\n' << key << " not found" << endl;
36
37     return 0;
38 |
39
40 //Binary search
41 int binarySearch( const int b[], int searchKey, int low, int high,
42                 int size )
43 |
44     int middle;
45
46     while ( low <= high ) {
47         middle = ( low + high ) / 2;
48
49         printRow( b, low, middle, high, size );
50
51         if ( searchKey == b[ middle ] ) //match
52             return middle;
53         else if ( searchKey < b[ middle ] )
```

```

54     high = middle - 1;          //search low end of array
55     else
56         low = middle + 1;        //search high end of array
57 }
58
59 return -1; //searchKey not found
60 |
61
62 //Print a header for the output
63 void printHeader( int size )
64 {
65     int i;
66
67     cout << "\nSubscripts;\n";
68
69     for ( i = 0; i < size; i ++ )
70         cout << setw( 3 ) << i << " ";
71
72     cout << '\n';
73
74     for ( i = 1; i <= 4 * size; i ++ )
75         cout << '-';
76
77     cout << endl;
78 }
79
80 //Print one row of output showing the current
81 //part of the array being processed.
82 void printRow( const int b[], int low, int mid, int high, int size )
83 {
84     for ( int i = 0; i < size; i ++ )
85         if ( i < low || i > high )
86             cout << " ";
87         else if ( i == mid )          //mark middle value
88             cout << setw( 3 ) << b[ i ] << '*';
89         else
90             cout << setw( 3 ) << b[ i ] << " ";
91
92     cout << endl;
93 }

```

输出结果:

Enter a number between 0 and 28; 25

Subscripts;

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|

-----

|   |   |   |   |   |    |    |     |    |    |    |    |    |    |    |
|---|---|---|---|---|----|----|-----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14* | 16 | 18 | 20 | 22 | 24 | 26 | 28 |
|---|---|---|---|---|----|----|-----|----|----|----|----|----|----|----|

25 not found



```

Enter a number between 0 and 28; 8
Subscripts:
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
-----
 0   2   4   6   8  10  12  14 * 16  18  20  22  24  26  28
 0   2   4   6 * 8  10  12
                8  10 * 12
                8 *
8 found in array element 4

Enter a number between 0 and 28; 6
Subscripts:
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
-----
 0   2   4   6   8  10  12  14 * 16  18  20  22  24  26  28
 0   2   4   6 * 8  10  12
6 found in array element 3

```

图 4.20 对排序后的数组进行二元搜索

## 4.9 多下标数组

C++ 中的数组可以有多个下标。多下标数组的常见用途之一便是表示值表格,它由排列成行和列的信息构成。为标识一个特定的表格元素,必须指定两个下标:第一个(按约定)指定元素所在的行,第二个(按约定)指定元素所在的列。

根据两个下标才能标识一个特定元素的表格或数组叫做双下标数组。注意,多下标数组也可能不止两个下标。事实上,C++ 编译器支持至少 12 个数组下标。图 4.21 展示了一个双下标数组 *a*。该数组包含 3 行和 4 列,所以我们说它是一个  $3 \times 4$  数组。通常,含有 *m* 行、*n* 列的一个数组就叫做一个  $m \times n$  数组。

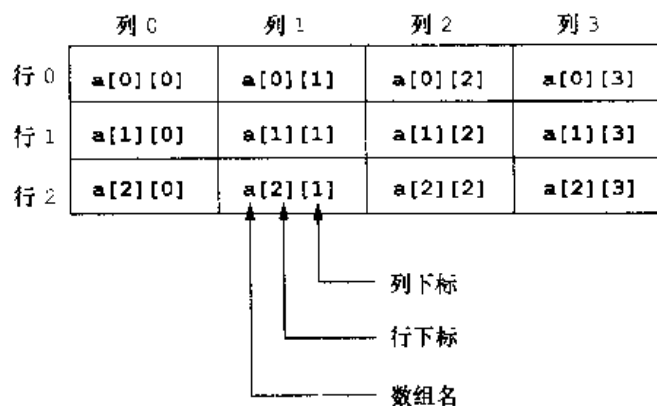


图 4.21 含有 3 行和 4 列的一个双下标数组

如图 4.21 所示的数组 *a* 中,每个元素都是用一个元素名来标识的,形如 *a*[*i*][*j*]。其中,*a* 是数组名,*i* 和 *j* 是下标,它们独一无二地标识了 *a* 中的每个元素。注意,对第 1 行的元素名来说,第一个下标肯定是 0;第 4 列的元素名中,第二个下标肯定是 3。

**常见编程错误 4.11** 将双下标数组元素 `a[ x ][ y ]` 写成 `a[ x, y ]` 错误的。实际上, `a[ x, y ]` 会被视为 `a[ y ]`, 因为 C++ 会对表达式(包含一个逗号操作符)进行求值, 所以 `x, y` 就相当于 `y`(取逗号分隔的表达式的一个)。

对一个多下标数组来说, 在其声明中, 可采取同单下标数组差不多的方式进行初始化。例如, 一个双下标数组 `b[ 2 ][ 2 ]` 可声明和初始化为

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

注意值是用花括号分组的。所以, 1 和 2 负责初始化 `b[ 0 ][ 0 ]` 和 `b[ 0 ][ 1 ]`; 而 3 和 4 负责初始化 `b[ 1 ][ 0 ]` 和 `b[ 1 ][ 1 ]`。假如在一行上没有给定足够多的初始化值, 那一行剩下的值就会自动初始化为 0。所以, 声明

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

会将 `b[ 0 ][ 0 ]` 初始化为 1, `b[ 0 ][ 1 ]` 初始化为 0, `b[ 1 ][ 0 ]` 初始化为 3, 而 `b[ 1 ][ 1 ]` 初始化为 4。

图 4.22 展示了如何在声明中对双下标数组进行初始化。程序声明了 3 个数组, 每个都有 2 行 3 列。array1 的声明用两个子列表提供了 6 个初始化值, 第一个子列表将数组的第 1 行初始化为值 1, 2, 3; 第二个子列表将数组的第 2 行初始化为值 4, 5, 6。若从 array1 的初始化列表中删除每个子列表两端的花括号, 编译器会自动先初始化第一行元素, 再初始化第二行元素。

```
1 //Fig. 4.22: fig04_22.cpp
2 //initializing multidimensional arrays
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void printArray( int [][ 3 ] );
9
10 int main()
11 {
12     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
13         array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
14         array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
15
16     cout << "Values in array1 by row are: " << endl;
17     printArray( array1 );
18
19     cout << "Values in array2 by row are: " << endl;
20     printArray( array2 );
21
22     cout << "Values in array3 by row are: " << endl;
23     printArray( array3 );
24
25     return 0;
26 }
27
```

```

28 void printArray( int a[][ 3 ] )
29 |
30     for ( int i = 0; i < 2; i ++ ) {
31
32         for ( int j = 0; j < 3; j ++ )
33             cout << a[ i ][ j ] << " ";
34
35         cout << endl;
36     }
37 }

```

输出结果:

```

Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0

```

图 4.22 初始化多维数组

array2 的声明提供了 5 个初始化值。这些初始化值会赋给第一行,然后赋给第 2 行。任何没有明确指定初始化值的元素都会自动初始化为零,所以 array2[ 1 ][ 2 ] 会被初始化为零。

array3 的声明用两个子列表提供了 3 个初始化值。用于第一行的子列表显式将第一行的头两个元素初始化为 1 和 2。第 3 个元素会自动初始化为零。用于第二行的子列表将第一个元素显式地初始化为 4。最后两个元素则自动初始化为零。

程序会调用函数 printArray,以输出每个数组的元素。注意,函数定义将数组参数指定为 int a[ ][ 3 ]。如果一个函数接受单下标数组作为其实参,那么在函数参数列表中,数组的方括号是空的。多下标数组的第一个下标的长度也是不需要的,但后续所有下标长度都是必需的。编译器利用这些长度判断多下标数组中的元素在内存中的位置。所有数组元素在内存中都是连续存放的,不管下标到底有多少。在一个双下标数组中,第一行先保存在内存中,紧接着保存第二行。

在参数声明中提供下标值,可使编译器指示函数如何定位数组中的一个元素。在一个双下标数组中,每一行都是一个单下标数组。为定位某个特定行中的元素,函数必须准确地知道每一行有多少个元素,这样便能在访问数组时,跳过正确数量的内存位置。所以,在访问 a[ 1 ][ 2 ] 时,函数知道跳过在内存中第一行的 3 个元素,从而来到第二行(行 1)。然后,函数会访问那一行的第 3 个元素(元素 2)。

许多常见的数据操作都用到了 for 重复结构。例如,for 结构

```

for ( column = 0; column < 4; column ++ )
    a[ 2 ][ column ] = 0;

```

可将数组 a(如图 4.21 所示)的第 3 行的全部元素设为 0。我们指定的是第 3 行,所以知道

第一个下标肯定是2(0是第一行下标,1是第二行下标)。for循环只对第二个下标(列下标)进行更改。上述for结构其实等价于以下语句

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

### 嵌套 for 结构

```
total = 0;

for ( row = 0; row < 3; row ++ )
    for ( column = 0; column < 4; column ++ )
        total += a[ row ][ column ];
```

可对数组a中所有元素进行求和。for结构以每次一行的方式,计算出数组元素的总和。外层for结构首先将row(行下标)设为0,使第一行的元素可由内层for结构进行求和。外层for结构使row增至1,使第二行元素能够求和。然后,外层for结构将row增至2,使第3行元素能够求和。嵌套的for结构中止后,会打印出结果。

图4.23中的程序用于对一个3×4的studentGrades数组执行其他常见的数组操作。数组每一行都代表一名学生,每一列都代表学生在本学期参加的4门考试之一的成绩。函数minimum决定了所有学生在本学期的最低成绩,函数maximum决定了任何学生在本学期的最高成绩,函数average决定了一名特定学生的学期平均成绩,函数printArray则采取一种整洁的、表格化的格式,输出双下标数组。

```
1 //Fig. 4.23: fig04_23.cpp
2 //Double-subscripted array example
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8
9 #include <iomanip>
10
11 using std::setw;
12 using std::setiosflags;
13 using std::setprecision;
14
15 const int students = 3; //number of students
16 const int exams = 4; //number of exams
17
18 int minimum( int [][ exams ], int, int );
19 int maximum( int [][ exams ], int, int );
20 double average( int [ ], int );
21 void printArray( int [][ exams ], int, int );
22
23 int main()
24 {
```

```
25 int studentGrades[ students ][ exams ] =
26     | | 77, 68, 86, 73 |,
27     | 96, 87, 89, 78 |,
28     | 70, 90, 86, 81 | |;
29
30 cout << "The array is;\n";
31 printArray( studentGrades, students, exams );
32 cout << "\n\nLowest grade: "
33     << minimum( studentGrades, students, exams )
34     << "\nHighest grade: "
35     << maximum( studentGrades, students, exams ) << '\n';
36
37 for ( int person = 0; person < students; person ++ )
38     cout << "The average grade for student " << person << " is "
39         << setiosflags( ios::fixed | ios::showpoint )
40         << setprecision( 2 )
41         << average( studentGrades[ person ], exams ) << endl;
42
43 return 0;
44 }
45
46 //Find the minimum grade
47 int minimum( int grades[][ exams ], int pupils, int tests )
48 |
49     int lowGrade = 100;
50
51     for ( int i = 0; i < pupils; i ++ )
52
53         for ( int j = 0; j < tests; j ++ )
54
55             if ( grades[ i ][ j ] < lowGrade )
56                 lowGrade = grades[ i ][ j ];
57
58     return lowGrade;
59 |
60
61 //Find the maximum grade
62 int maximum( int grades[][ exams ], int pupils, int tests )
63 |
64     int highGrade = 0;
65
66     for ( int i = 0; i < pupils; i ++ )
67
68         for ( int j = 0; j < tests; j ++ )
69
70             if ( grades[ i ][ j ] > highGrade )
71                 highGrade = grades[ i ][ j ];
72
73     return highGrade;
74 |
75
```

```

76 //Determine the average grade for a particular student
77 double average( int setOfGrades[], int tests )
78 {
79     int total = 0;
80
81     for ( int i = 0; i < tests; i ++ )
82         total += setOfGrades[ i ];
83
84     return static_cast < double >( total ) / tests;
85 }
86
87 //Print the array
88 void printArray( int grades[][ exams ], int pupils, int tests )
89 {
90     cout << "           [0] [1] [2] [3] ";
91
92     for ( int i = 0; i < pupils; i ++ ) {
93         cout << "\nstudentGrades[" << i << "]" << " ";
94
95         for ( int j = 0; j < tests; j ++ )
96             cout << setw( 5 ) << grades[ i ][ j ];
97     }
98 }
99 }

```

输出结果:

The array is:

|                  | [1] | [2] | [3] | [4] |
|------------------|-----|-----|-----|-----|
| studentGrades[0] | 77  | 68  | 86  | 73  |
| studentGrades[1] | 96  | 87  | 89  | 78  |
| studentGrades[2] | 70  | 90  | 86  | 81  |

Lowest grade: 68

Highest grade: 96

The average grade for student 0 is 76.00

The average grade for student 1 is 87.50

The average grade for student 1 is 81.75

图 4.23 双下标数组用法示例

函数 `minimum`, `maximum` 和 `printArray` 各自都接受 3 个实参——`studentGrades` 数组(在每个函数中叫做 `grades`), 学生数量(数组行数)以及考试数(数组列数)。每个函数都会用嵌套的 `for` 结构遍历数组 `grades`。以下嵌套的 `for` 结构

```

    for ( int i = 0; i < pupils; i ++ )
        for ( int j = 0; j < tests; j ++ )
            if ( grades[ i ][ j ] < lowGrade )
                lowGrade = grades[ i ][ j ];

```

是从函数 `minimum` 的定义中摘录的。外层 `for` 结构首先将 `i`(行下标)设为 0, 所以第一行的元素可在内层 `for` 结构主体中, 同变量 `lowGrade` 进行比较。内层 `for` 结构会遍历一个特定行的 4

个成绩,并将每个成绩同 lowGrade 比较。如成绩小于 lowGrade,lowGrade 就设成那个成绩。然后,外层 for 结构将行下标增至 1,第二行的元素会同变量 lowGrade 进行比较。然后外层 for 结构将行下标增至 2,第三行的元素又会同变量 lowGrade 进行比较。当嵌套结构执行完毕后,lowGrade 将包含双下标数组中最小的成绩。函数 maximum 的工作方式同函数 minimum 类似。

函数 average 会取得两个实参:一个单下标数组,其中包含了特定学生的考试结果,以及一个数组中的考试结果的数量。调用 average 时,第一个实参是 studentGrades[ student ],它指定双下标数组 studentGrades 的一个特定行传递给 average。例如,实参 studentGrades[ 1 ] 代表保存在双下标数组 studentGrades 的第二行中的 4 个值(由成绩构成的一个单下标数组)。可将一个双下标数组想象成它的元素都是单下标的数组。函数 average 将计算数组元素的总和,将结果除以考试结果的数量,并返回浮点结果。

## 4.10 【可选案例分析】对象思想:标识类的行为

在第 2 章和第 3 章的“对象思想”小节中,我们完成了电梯模拟程序的面向对象设计的前几个步骤。在第 2 章,我们标识了需要实现的几个类,并创建了一个类图,它对系统结构进行了建模处理。在第 3 章,我们还决定了类的许多属性,我们对 Elevator 类可能的状态进行了调查,并用状态图来表示它们。另外,还用活动图对电梯用于响应按钮操作的逻辑进行了建模。

本节的重点是判断类的操作(或行为),它们是实现电梯模拟程序所必需的。在第 5 章,还会强调类对象之间的合作(交互)。

类的行为其实就是一种服务,由类提供给那个类的“客户”。让我们来考虑一些现实世界的类的行为情况。对一台收音机来说,它的行为包括设置电台和音量(通常由用户调节收音机的按钮来完成)。一辆车的操作包括加速(由踩下加速踏板来完成)、减速(由踩下刹车踏板来完成)和换挡等等。

对象通常不会自发地执行它们的行为。相反,只有某个发送对象(通常叫做客户对象)向接收对象(通常叫做服务器对象)发送一条消息,请求那个接收对象执行一项特定的行为,那项行为才会执行。这听起来就像一个成员函数调用——准确地说,类似于在 C++ 中将消息发送给对象。本节我们将探讨在电梯模拟系统中,我们的类应该将哪些行为提供给它们的“客户代码”。

每个类的许多行为都可直接从问题陈述中派生出来,为此,需要在问题陈述中对动词和动词短语进行检查。然后,可将这些短语同系统中一个特定的类关联起来(参见图 4.24)。在图 4.24 中,许多动词短语都可帮助我们判断类需要采取的行为。

| 类         | 动词短语                                        |
|-----------|---------------------------------------------|
| Elevator  | 移动,抵达一个楼层,重置电梯按钮,电梯响铃,通知抵达一个楼层,开门,关门        |
| Clock     | 每秒滴答一次                                      |
| Scheduler | 随机安排时间,创建一个人,指示一个人走到楼层,检查一个楼层是否有人,推迟一秒创建一个人 |
| Person    | 走到楼层,按楼层按钮,按电梯按钮,走进电梯,离开电梯                  |
| Floor     | 重置电梯按钮,关灯,开灯                                |

(续表)

| 类              | 动词短语                        |
|----------------|-----------------------------|
| FloorButton    | 召唤电梯                        |
| ElevatorButton | 通知电梯移动                      |
| Door           | (门已打开)通知人离开电梯,(门已打开)通知人进入电梯 |
| Bell           | 问题陈述中没有动词短语                 |
| Light          | 问题陈述中没有动词短语                 |
| Building       | 问题陈述中没有动词短语                 |

图 4.24 模拟系统中每个类的动词短语

为了根据这些动词短语创建不同的行为,让我们依次检查每个类列出的动词短语。随 Elevator 类列出的动词“移动”是指电梯在楼层之间的移动。那么,“移动”应该成为 Elevator 的一种行为吗?并没有什么消息指示电梯移动;相反,电梯之所以移动,是为了响应在门已关上的前提下的一次按钮行为。所以,“移动”并不对应一项行为。“抵达楼层”这个短语也不算一种行为,因为电梯本身决定了何时抵达楼层——根据的是时间。

“重置电梯按钮”短语暗示着电梯要向电梯按钮发送一条消息,指示按钮进行重置。所以,ElevatorButton 需要一个行为,以便为电梯提供这种服务。在我们的类图中(图 4.25),我们将这个行为放在 ElevatorButton 类的底部。行为的名称表示为函数名,并提供与返回类型有关的信息

```
resetButton():void
```



图 4.25 含有属性和操作的类图

首先写好行为名称,然后是一对圆括号,其中包含一个用逗号分隔的参数列表,以便提



供给该行为(目前无任何参数)。在参数列表之后,是一个冒号,然后是该行为的返回类型(目前是 `void`)。注意,我们的大多数行为都是没有参数的,而且返回类型为 `void`。随着设计和实现过程的继续,这也有可能发生改变。

从 `Elevator` 类列出的“电梯响铃”短语中,我们总结出 `Bell` 类应当有一个行为来提供响铃服务,所以在 `Bell` 类下面列出 `ringBell` 操作。

电梯抵达一个楼层后,它会“通知抵达一个楼层”,而楼层的响应是采取多种行动(即重置楼层按钮,并开灯),所以, `Floor` 类需要一个行为来提供这种服务。我们把这个行为命名为 `elevatorArrived`,并在图 4.25 中,将行为名放在 `Floor` 类的底部。

`Elevator` 类剩下的两个动词短语指出,电梯需要开门和关门,所以, `Door` 类需要提供这些行为。我们在 `Door` 类的底部列出 `openDoor` 和 `closeDoor` 这两个行为。

`Clock` 类列出的短语是“每秒滴答一次”,这个短语为我们带来了一个有趣的问题。显然,“获取时间”是由时钟提供的一个行为,但时钟滴答也是一种行为?为解答这个问题,让我们关注一下模拟系统的工作原理。

在问题陈述中指出,调度器(`Scheduler`)需要知道当前时间,以决定是否创建一个新人,并令其走到一个楼层。电梯需要时间来判断何时抵达一个楼层。我们还决定,建筑物需要负责运行模拟系统,并将时间传给调度器和电梯。现在,来看看模拟系统是如何运行的。在模拟系统运行的每一秒时间内,建筑物都会重复以下步骤:

- (1) 从时钟取得时间。
- (2) 向调度器提供时间,使调度器能在必要时创建一个新人。
- (3) 向电梯提供时间,使电梯能判断是否抵达了一个楼层(如果电梯正在移动的话)。

我们决定,建筑物需要全权负责运行模拟系统的所有部分。所以,建筑物还必须能够对时钟进行增值,每秒钟,时钟应该增值一次;然后,时间应该传递给调度器和电梯。

由此便造成了两个行为 `getTime` 和 `tick`,它们都列于 `Clock` 类之下,其中, `getTime` 行为以一个 `int` 值的形式,返回时钟的时间属性。在上述列表的第 2 项和第 3 项中,我们看到了这样的短语:“向调度器提供时间”和“向电梯提供时间”。所以,可以为 `Scheduler` 和 `Elevator` 类提供 `processTime` 行为,还可为 `Building` 类添加 `runSimulation` 行为。

`Scheduler` 类列出了这两个动词短语“随机安排时间”和“推迟一秒创建一个人”。调度器决定由自己来执行这些行为,所以不需要将这些服务提供给客户。因此,上述两个动词短语不对应任何行为。

`Scheduler` 类列出的“创建一个人”短语则展示了一种特殊情况。尽管我们可为 `Scheduler` 类建模一个对象,令其发送一个“创建”消息,但 `Person` 类的一个对象却不可响应这样的“创建”消息,因为那个对象尚不存在。对象的创建是实现细节要负责的事儿,所以不能表示成一个类的行为。等我们在第 7 章讨论实现时,还会继续讨论新对象的创建问题。

图 4.24 还列出了“指示一个人走到楼层”这个短语,它的意思是, `Person` 类应该有一个操作可由调度器调用,以便告诉那个人走到一个楼层。我们把这个行为叫做 `stepOntoFloor`,并在 `Person` 类的下面列出它。

“检查楼层是否有人”这个短语暗示着, `Floor` 类需要提供一项服务,允许系统中的对象知道楼层当前是被人占(`Occupied`),还是未被人占。为此服务创建的行为应该在楼层被人

占据的情况下返回 true,而在无人的情况下返回 false。所以在 Floor 类的底部,我们设置了这样的一个行为

```
isOccupied() : bool
```

Person 类列出了“按楼层按钮”和“按电梯按钮”这两个短语。所以,我们在 UML 类图(图 4.25)中,在 FloorButton 和 ElevatorButton 类下,列出了 pressButton 操作。注意,在分析 Scheduler 类的动词短语时,已经解释过了一个人“走到楼层”的事实,所以不再需要根据随 Person 类列出的“走到楼层”短语而创建任何操作。随 Person 类列出的“进入电梯”和“离开电梯”这两个短语暗示着 Elevator 类需要采取同这些行动对应的操作<sup>①</sup>。

Floor 类还列出了“重置楼层按钮”这个短语,所以在 FloorButton 类之下,我们列出了相应的 resetButton 行为。Floor 类还列出了“关灯”和“开灯”这两个短语,所以我们会创建 turnOff 和 turnOn 这两个行为,并把它列于 Light 类之下。

在 FloorButton 类下而列出的“召唤电梯”短语暗示着, Elevator 类需要一项 summonElevator 行为。ElevatorButton 类列出的“通知电梯移动”短语则暗示着 Elevator 类需要提供一项“移动”服务。但是,在电梯能正常移动之前,必须先关好门。所以,更恰当的一种做法是,在 Elevator 类之下列出一个 prepareToLeave 行为,以便电梯在移动之前,完成一系列必要的准备工作。

Door 类列出的短语暗示着门会向一个人发送消息,指示他离开电梯或进入电梯。所以,我们为 Person 类创建了两个行为,分别对应这两种行为——exitElevator 和 enterElevator。

到目前为止,我们并没有过多地关心参数或返回类型的情况;我们惟一的宗旨只是对于每个类的行为有一个基本的理解。随着设计过程的深入,每个类的行为数量还有可能发生变化——既可能发现有新的行为需要添加,也可能发现当前的行为不再需要。

## 顺序图

可用 UML 顺序图(参见图 4.26)来建模我们的“模拟循环”——亦即在前面的讨论中,建筑物在模拟系统运行期间,需要重复采取的步骤。顺序图的重点在于随着时间的推移,消息如何在不同的对象之间传递。

每个对象都用图中位于顶部的一个矩形表示,对象名放在矩形内部。我们利用第 2 章末“对象思想”小节的对象图中介绍的约定,来书写顺序图中的对象名(参见图 2.45)。从对象矩形向下延展的虚线是对象的生命线,生命线代表时间的推移过程。沿对象生命线排列的一系列行动是从上到下,按时间顺序发生的——靠近生命线顶部的行动要先于靠近底部的行动发生。

在顺序图中,两个对象之间的消息是用一条实箭头线表示的,从发出消息的对象,一直延展到负责接收的对象。消息会在负责接收的对象中,调用相应的操作。箭头指向负责接收消息的那个对象的生命线。消息名称会出现在消息线的上方,而且应包括要传递的任何参数。例如, Building 类的对象会向 Elevator 类的对象发送 processTime 消息。因此,这个消

<sup>①</sup> 目前,我们只能猜测这些操作具体是如何采取的。例如,在对真正的电梯进行建模时,也许会采用一个传感器,用它检测乘客进入和离开电梯的情况。就目前来说,只需列出这些操作就可以了。等将重点转移到用 C++ 来实现模拟程序时,就知道这些操作具体采取的行动了。

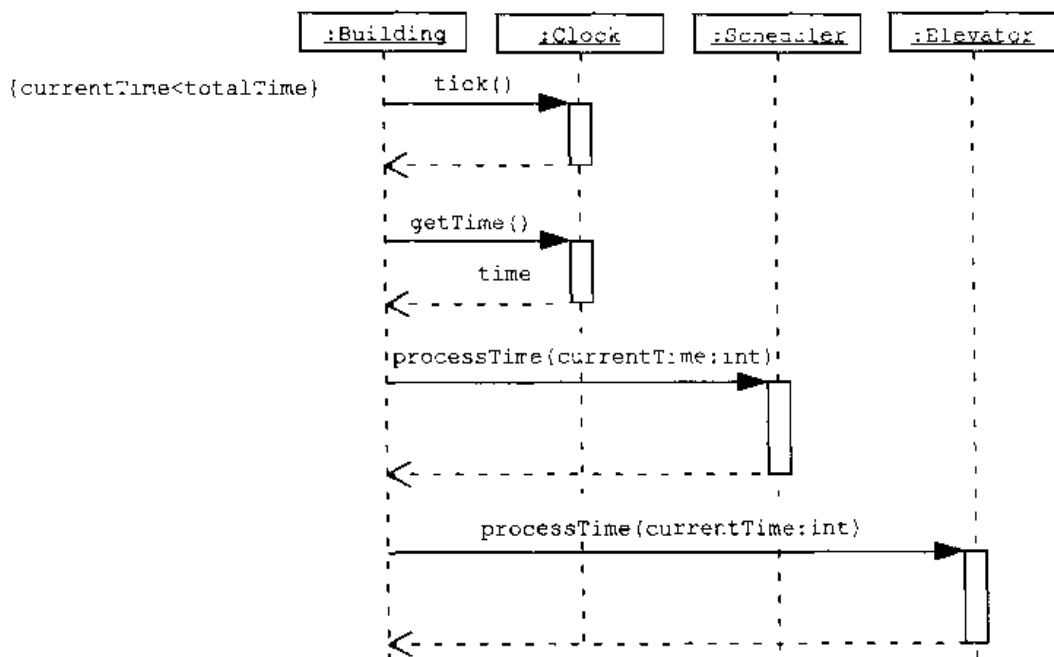


图 4.26 用于对模拟循环进行建模的顺序图

息名应出现在消息线上方,而参数名(currentTime)应出现在消息名右侧的圆括号中;每个参数名后面都应跟随一个冒号,再加上参数的类型。

假如一个对象会返回控制流程,或者返回一个值,那么应该返回一条消息(表示成虚箭头线),从负责返回控制权的对象延展到当初发出消息的对象。例如,Clock 类的对象会返回 time,以响应从 Building 类的对象中接收到的 getTime 消息。

沿着对象生命线摆放的矩形(名为激活—activations)各自代表一种活动的持续时间。当对象接收到一条消息时,就会初始化一个“激活”,这是用那个对象的生命线上的一个矩形来表示的。矩形的高度对应于由消息初始化的活动(可能有多项活动)——活动持续时间越长,矩形越高。

在图 4.26 中,最左边的文本指定的是一个计时限制。在当前时间小于总计模拟时间的情况下(currentTime < totalTime),对象就会根据图中建模的顺序,不断地将消息发送给另一个对象。

图 4.27 展示的是调度器如何处理时间,以及如何创建新人,以便走到楼层。针对这个图,我们假定在与建筑物提供的时间相符的一个时刻,调度器已安排好一个人走到两个楼层上。让我们通过这个顺序图,跟随消息的流程走一遍。

对象 building 首先将 processTime 消息发送给 scheduler,并传送当前时间。然后,scheduler 对象必须决定是否新建一个人,以便走到第一个楼层(由 Floor 类的 floor1 对象表示)。问题陈述告诉我们,调度器必须先验证楼层未被人占据,才能新建一个人,令其走到那个楼层。所以,scheduler 对象必须向 floor1 对象发送一条 isOccupied 消息,以完成这个任务。

floor1 对象要么返回 true,要么返回 false(由消息返回虚线和 bool 类型注明)。在这个时候,scheduler 对象的生命线会分割成两条并行的生命线,以代表对象可能选择的每个可能的消息发送顺序——具体由 floor1 对象的返回值决定。对象的生命线可分割成两条或更多的

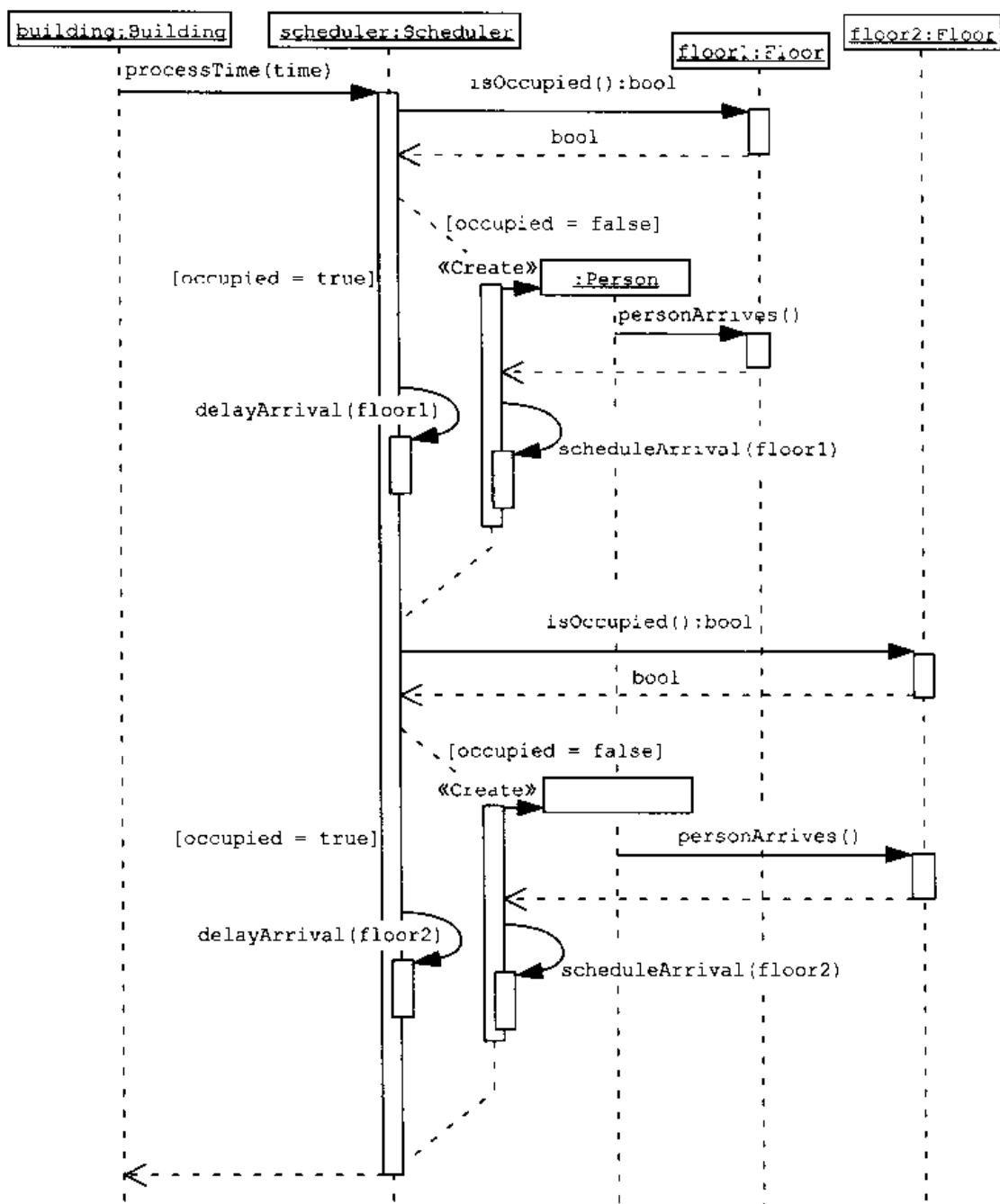


图 4.27 调度过程的顺序图

生命线,以便标识活动的条件执行。必须为每条生命线提供一个条件,新生命线与主生命线并行,同时生命线可能在以后某个时候会交叉。

假如 floor1 对象返回 true(表明楼层被占据),scheduler 就会调用它自己的 delayArrival 函数,传递一个参数,指出 floor1 到达时间需要重新安排。这个函数并不是 Scheduler 类的一种操作,因为其他对象是不会调用它的。delayArrival 函数只是 Scheduler 在一个操作内部采取的行动。注意,当 scheduler 对象向自己发送一条消息时(即调用它自己的一个成员函数),那条消息的激活矩形会在当前激活矩形的右侧居中显示。

假如 floor1 对象返回 false(表明楼层未被占据),那么 scheduler 对象会为 Person 类新建

一个对象。在顺序图中,当新对象创建时,新对象的矩形会置于同对象创建时间对应的一个垂直位置。负责创建另一个对象的对象会发送一条消息,其中含有“create”字样,并将其封闭在一对书名号中(《 》)。这条消息的箭头指向新对象的矩形。对象生命期结束时的一个大“X”指出那个对象被删除的时刻(注意:我们的顺序图没有标记 Person 类的任何对象的删除时刻,所以,图中没有出现“X”标记。将在第 7 章讨论如何利用 C++ 的 new 和 delete 操作符,来动态创建和删除对象)。

创建 Person 类的新对象后,相应的人必须接着走到第一层。所以,新的 Person 对象会向 floor1 对象发送一条 personArrives 消息。该消息通知 floor1 对象,已经有一人走入楼层。

scheduler 对象已创建 Person 类的一个新对象,它会为 floor1 安排一次新的到达(一个新人进入楼层)。scheduler 对象会调用它自己的 scheduleArrival 函数,而这个调用的激活矩形会在当前激活矩形的右侧居中显示。scheduleArrival 函数不是一项操作,它只是由 Scheduler 类在一个操作内部采取的行动。届时,两条生命线会相交(会聚)。然后,scheduler 对象会按照与第一层相同的方式,对第二个楼层进行处理。scheduler 完成了对 floor2 的处理后,scheduler 对象会将控制权返回给 building 对象。

本节讨论了类的操作,并介绍了 UML 顺序图,以便图示这些操作。在第 5 章的“对象思想”小节中,还会探讨一个系统中的对象相互间如何沟通,以完成特定的任务,并开始用 C++ 来实现我们的电梯模拟程序。

## 4.11 小结

- C++ 用数组保存值列表。数组是一组连续的相关内存位置,这些位置的相互关系在于它们都有相同的名称和相同的类型。为了对数组中一个特定的位置或元素进行引用,我们需要指定数组名和下标。下标指出从数组开头算起的元素个数。
- 下标必须为整数,或为整型表达式。下标表达式会先进行求值,以决定数组中的特定元素。
- 必须分清“数组第 7 个元素”和“数组元素 7”的差异。第 7 个元素的下标是 6,而数组元素 7 的下标是 7(实际是数组的第 8 个元素)。如混淆这一概念,容易造成“相差 1”错误。
- 数组会占用内存空间。为了为整型数组 b 保留 100 个元素,并为整型数组 x 保留 27 个元素,程序员需要这样写  

```
int b[ 100 ], x[ 27 ];
```
- char 类型的数组可用于保存字符串。
- 可通过声明、赋值或输入,完成对数组元素的初始化。
- 假如初始化值的数量少于数组元素,剩余的元素会自动初始化为零。
- C++ 并不禁止引用超出数组边界的元素。
- 可用字符串字面值来初始化字符数组。
- 所有字符串都用 NULL 字符结束(‘\0’)。
- 可在初始化列表中,用字符常量对字符数组进行初始化。
- 保存在数组中的一个字符串的单独元素可使用数组下标直接访问。

- 为了将一个数组传给函数,需要传递数组名称。为了将数组的单个元素传给函数,只需传递数组名称,并在后面跟上那个特定元素的下标(包含在一个对方括号中)即可。
- 数组传给函数时,采用的是模拟的“按引用调用”——被调用的函数能修改调用者原始数组中的元素值。数组名的值就是数组第一个元素的地址。由于数组的起始地址已经传递,所以被调用的函数准确知道数组存放于何处。
- 为接收一个数组实参,函数的参数列表必须指出即将接收的是一个数组。对于单下标数组参数来说,无需在方括号中加上数组的长度。
- C++ 提供了类型限定符 `const`,可用于防止在函数中对数组进行修改。假如一个函数在指定一个数组参数的同时,在它前面加了一个 `const` 限定符,那么数组元素会在函数体内成为常量。以后,任何在函数体中对数组元素进行修改的企图都会导致一个语法错误。
- 一个数组可采取冒泡排序技术进行排序。期间,需要采取多个步骤,每个步骤都会依次比较不同的连续元素对。假如一对元素本身的顺序是对的(或者两个值相等),那么保持不变;如果一对元素本身的顺序是错的,则交换两个值的位置。对于小数组来说,冒泡排序是可以接受的。但对较大的数组来说,它的效率没有其他较复杂的排序算法高。
- 线性搜索会用“搜索键”对数组的每个元素进行比较。假如数组本身未采取任何特定的顺序,那么目标值既有可能在第一个元素中找到,也有可能最后一个元素中找到。所以,平均地算下来,程序必须拿搜索键同数组的一半元素进行比较。线性搜索方法特别适合小型数组;而且对于未排好序的数组来说,也是能够接受的。
- 二元搜索会找到数组的中间元素,并用它同搜索键进行比较。所以每一次比较,都可以排除数组中一半的元素。如两者相等,表明找到了搜索键,并返回那个元素的下标。否则,又会针对数组的另一半进行搜索。
- 采用二元搜索,在最坏的情况下,针对一个 1 024 个元素的数组,最多只需进行 10 次比较。
- 可用数组表示值表格,其中的信息排列成行和列的形式。为了标识表格中的一个特定的元素,需要指定两个下标,第一个(按约定)指定元素所在的行,第二个(按约定)指定元素所在的列。根据两个下标才能标识一个特定元素的表格或数组叫做双下标数组。
- 如果一个函数接受单下标数组作为其实参,那么在函数参数列表中,数组的方括号是空的。多下标数组的第一个下标的长度也是不需要的,但后续所有下标长度都是必需的。编译器利用这些长度判断多下标数组元素在内存中的位置。
- 为了将双下标数组的一行传给接收单下标数组的函数,只需传递那个数组的名称,并在后面跟上第一个下标。

## 本章术语

array 数组

array initializer list 数组初始化列表

binary search of an array 对数组进行二元搜索

bounds checking 边界检查

bubble sort 冒泡排序  
 column subscript 列下标  
 constant variable 常量变量  
 const type qualifier const 类型限定符  
 declare an array 声明一个数组  
 double-subscripted array 双下标数组  
 element of an array 数组元素  
 initialize an array 初始化一个数组  
 linear search of an array 对数组进行线性搜索  
 magic number 魔数  
 m-by-n array  $m \times n$  数组  
 multiple-subscripted array 多下标数组  
 name of an array 数组名  
 named constant 已命名常量  
 null character( '\0' ) 空字符( '\0' )  
 off-by-one error “相差1”错误  
 passing arrays to functions 向函数传数组  
 pass-by-reference 引用传递  
 pass of a bubble sort 冒泡排序的步骤  
 position number 位置编号

row subscript 行下标  
 scalability 伸缩性  
 scalar 标量  
 search an array 搜索一个数组  
 search key 搜索键  
 simulated call-by-reference 模拟的“按引用调用”  
 single-subscripted array 单下标数组  
 sinking sort 下沉排序  
 sort an array 对一个数组进行排序  
 square brackets [ ] 方括号 [ ]  
 string 字符串  
 subscript 下标  
 table of values 值表格  
 tabular format 表格式数据  
 temporary area for exchange of values  
 交换值的临时区域  
 tripe-subscripted array 三下标数组  
 value of an element 元素值  
 “walk off” an array 遍历一个数组  
 zeroth element 第零个元素

## “对象思想”术语

activation rectangle symbol in UML sequence diagram  
 UML 顺序图中的“激活矩形”符号  
 behavior 行为  
 client object 客户对象  
 collaboration 合作  
 conditional execution of activities 活动的条件执行  
 duration of activity 活动持续时间  
 flow of messages in UML sequence diagram  
 UML 顺序图中的消息流程  
 guillemets 书名号  
 line with solid arrowhead in sequence diagram  
 顺序图中的实箭头线  
 message 消息  
 object lifeline in UML sequence diagram

UML 顺序图中的对象生命线  
 object rectangle symbol in UML sequence diagram  
 UML 顺序图中的对象矩形  
 operation 行为  
 return message symbol in UML sequence diagram  
 UML 顺序图中的返回消息符号  
 return type of an operation 行为返回类型  
 sequence diagram 顺序图  
 server object 服务器对象  
 service that an object provides 对象提供的服务  
 simulation loop 模拟循环  
 splitting an object's lifeline 分割对象的生命线  
 verbs in a problem statement 问题陈述中的动词

## 常见编程错误

- 4.1 有必要注意“数组第7个元素”和“数组元素7”的差异。由于数组下标自0开始,所以“数组第7个元素”的下标是6。相反,“数组元素7”的下标就是7,而且事实上是数组的第8个元素。令人遗憾的是,如混淆这两种提法,往往会造成“相差1”错误。
- 4.2 本应初始化的数组元素却忘了初始化是逻辑错误。
- 4.3 在数组初始化列表中提供的初始化值数量多于数组元素的数量是语法错误。

- 4.4 在可执行语句中向常量变量赋值,属于语法错误。
- 4.5 只有常量才可用于声明自动和静态数组。不用常量会造成语法错误。
- 4.6 如引用数组边界之外的元素,会造成运行时逻辑错误。但这不属于语法错误。
- 4.7 尽管可能在 for 循环和另一个嵌套于其中的 for 循环中使用相同的计数器变量,但这通常都会造成逻辑错误。
- 4.8 用 `cin >>` 输入数据时,若键盘向字符数组输入超出其容量的字符串,会造成程序中数据丢失,以及其他严重的运行错误。
- 4.9 假定每次调用函数时,它的局部 static 数组的元素都重新初始化为零,有可能导致程序中产生逻辑错误。
- 4.10 如忘了数组是按引用传递的(所以是完全能够修改的),有可能造成逻辑错误。
- 4.11 将双下标数组元素 `a[ x ][ y ]` 写成 `a[ x, y ]` 是错误的。实际上, `a[ x, y ]` 会被视为 `a[ y ]`,因为 C++ 会对表达式(包含一个逗号操作符)进行求值,所以 `x, y` 就相当于 `y` (取逗号分隔的表达式的一个)。

### 良好编程习惯

- 4.1 将数组的长度定义成常量变量,而非取字面值的常量,可使程序显得更清晰。利用这一技术,可有效地防范所谓的“魔数”;例如,针对包含 10 个元素的一个数组,假如在数组处理代码中,反复地提到长度 10,那么“10”这个数字就显得十分重要。一旦在程序中使用其他“10”,但它又和数组长度毫不相关,就会使读者混淆。
- 4.2 坚决保证程序的条理清晰!有时甚至宁愿不要最有效地利用内存或处理器时间,也要编写出更清晰的程序。
- 4.3 有的程序员在函数原型中包括变量名,目的是使程序更清晰。但编译器会忽略这些名称。

### 性能提示

- 4.1 假如不是用执行时的赋值语句来初始化数组,而是在编译时用一个数组初始化列表来初始化数组,程序执行速度会更快。
- 4.2 有时,对性能的考虑要优先于对程序清晰性的考虑。
- 4.3 可向局部数组声明应用 static 说明符,使数组不至于在每次调用函数时都进行创建和初始化。同时,当函数每次在程序中退出时,也不至于被删除。这样可提高性能。
- 4.4 通过模拟的“按引用调用”来传递数组,可在一定程度上改善性能。假如数组通过“传值调用”(call-by-value)进行传通,那么需要传通每个元素的一个副本。对于大型的、经常传递的数组来说,这显然既会浪费时间,又会浪费可观的存储空间来保存数组副本。
- 4.5 有时,最简单的算法在性能上也是最差的。它们惟一的优点便是容易编写、测试和调试。为了获得最好的性能,往往需要采取更复杂的算法。
- 4.6 二元搜索相较线性搜索所带来的巨大性能提升也不是没有代价的。对数组进行排序本身便是一个耗时耗力的工作,它比每次搜索一个项目要“费劲儿”得多。不过,假如需要不断地搜索一个数组,同时又想获得最好的性能,那么提前为它排好序仍是颇为值得的。



### 可移植性提示

- 4.1 如引用数组边界之外的元素,所造成的影响(通常都是严重的)与具体系统有关。这通常会修改一个无关变量的值。

### 软件工程知识

- 4.1 将每个数组的长度定义成常量变量,但非常量可改善程序的伸缩性。
- 4.2 完全有可能按值来传递一个数组(利用第 16 章解释的简单技巧即可)——尽管极少有人会这样做。
- 4.3 `const` 类型限定符可应用于函数定义中的数组参数,从而防止原始数组在函数体中被不慎修改。这是“最低权限”原则的另一个例子。除非绝对需要,否则不要为函数赋予对修改数组的权力。

### 测试和调试提示

- 4.1 遍历一个数组时,数组下标永远不能小于 0,而且永远都要小于数组中的元素总数(比数组长度小 1)。请在循环中止条件中,杜绝访问超出这个范围的元素。
- 4.2 程序应检查所有输入值的正确性,防止错误信息影响程序计算。
- 4.3 稍后学习类时(从第 6 章开始),你将知道如何开发一个“智能数组”,它能在运行时自动检查所有下标引用都在边界之内。使用这样的智能数据类型,有助于消除程序中的 Bug。
- 4.4 尽管可在 `for` 主体中修改循环计数器,但请尽量避免这样做,因为这通常都会导致容易被忽视的 Bug。

### 自测题

- 4.1 填空题:
- 列表和表格的值保存在\_\_\_\_\_中。
  - 数组元素之间的关系是它们具有同样的\_\_\_\_\_和\_\_\_\_\_。
  - 用来引用某个特定数组元素的数值被称为该数组的\_\_\_\_\_。
  - \_\_\_\_\_用于声明数组的长度,因为它能够使程序具有更强的伸缩性。
  - 按顺序排列数组元素的过程称为\_\_\_\_\_数组。
  - 判断数组中是否包含特定键值的过程称为\_\_\_\_\_数组。
  - 使用了两个下标的数组可以称作\_\_\_\_\_数组。
- 4.2 判断正误。如果不正确,请说明原因。
- 数组可以保存许多不同类型的值。
  - 数组下标通常为浮点类型。
  - 如果初始化列表中的初始化值少于数组元素的个数,那么剩余的元素就会被自动初始化为初始化值列表中的最后那个值。
  - 初始化列表中包含的初始化值超过数组元素的个数是不正确的。
  - 对于一个单一数组元素来说,如果它被传递给函数,并在函数中被修改,那么在被调用函数执行完成后,它会包含过修改后的值。
- 4.3 回答下列关于数组 `fractions` 的问题:

- a) 定义一个常量变量 `arraySize`, 并将其初始化为 0。
- b) 把 `arraySize` 数组元素声明为 `double` 类型, 并将元素初始化为 0。
- c) 为数组中的第 4 个元素命名。
- d) 引用数组元素 4。
- e) 把 1.667 这个值赋给数组元素 9。
- f) 把 3.333 这个值赋给数组中的第 7 个元素。
- g) 打印数组元素 6 和 9, 小数点后面为 2 位精度, 并实际在屏幕上显示输出结果。
- h) 用 `for` 重复结构打印数组的所有元素。把整数变量 `x` 定义为循环控制变量。显示结果输出。

#### 4.4 回答下列关于 `table` 数组的问题:

- a) 把数组声明为整数数组, 令其有 3 行 3 列。前提是常量变量 `arraySize` 已经定义为 3。
- b) 数组中可以包含多少元素?
- c) 用 `for` 重复结构把数组中的每个元素初始化为该元素下标的和。前提是整数变量 `x` 和 `y` 已被声明为控制变量。
- d) 编写一个程序片段, 用 3 行 3 列表格的形式打印数组表格中各个元素的值。前提是已用声明

```
int table[ arraySize ][ arraySize ] =
    { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

初始化该数值, 而且整数变量 `x` 和 `y` 已被声明为控制变量。显示输出结果。

#### 4.5 找出下列程序段中的错误, 并说明如何改正:

- a) `#include <iostream>;`
- b) `arraySize = 10; //arraySize was declared const`
- c) 假设: `int b[ 10 ] = { 0 };`  
`for ( int i = 0; i <= 10; i ++ )`  
`b[ i ] = 1;`
- d) 假设: `int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`  
`a[ 1, 1 ] = 5;`

### 自测题答案

- 4.1 a) 数组 b) 名称、类型 c) 下标 d) 常量变量 e) 排序 f) 查找 g) 双下标
- 4.2 a) 错误。数组中只能保存同种类型的数值。
- b) 错误。数组下标通常应该是整数或整数表达式。
- c) 错误。剩余的元素会被自动初始化为 0。
- d) 正确。
- e) 错误。数组中的单个元素通过传值调用来传递。如果整个数组都传递到函数中, 那么任何修改都会对原始数组产生影响。
- 4.3 a) `const int arraySize = 10;`
- b) `double fractions[ arraySize ] = { 0 };`
- c) `fractions[ 3 ]`

```

d) fractions[4]
e) fractions[9] = 1.667;
f) fractions[6] = 3.333;
g) cout << setiosflags( ios::fixed | ios::showpoint )
    << setprecision( 2 ) << fractions[ 6 ] << "
    << fractions[ 9 ] << endl;

```

输出结果:

```
3.33 1.67
```

```

h) for ( int x = 0; x < arraySize; x ++ )
    cout << "fractions[" << x << "] = " << fractions[ x ]
    << endl;

```

输出结果:

```

fractions[ 0 ] = 0
fractions[ 1 ] = 0
fractions[ 2 ] = 0
fractions[ 3 ] = 0
fractions[ 4 ] = 0
fractions[ 5 ] = 0
fractions[ 6 ] = 3.333
fractions[ 7 ] = 0
fractions[ 8 ] = 0
fractions[ 9 ] = 1.667

```

4.4 a) `int table[ arraySize ][ arraySize ];`

b) 9 个

```

c) for ( x = 0; x < arraySize; x ++ )
    for ( y = 0; y < arraySize; y ++ )
        table[ x ][ y ] = x + y;
d) cout << "    [0] [1] [2]" << endl;
    for ( int x = 0; x < arraySize; x ++ ) {
        cout << '[' << x << " ] ";
        for ( int y = 0; y < arraySize; y ++ )
            cout << setw( 3 ) << table[ x ][ y ] << " ";
        cout << endl;
    }

```

输出结果:

```

    [0] [1] [2]
[0]  1   8   0
[1]  2   4   6
[2]  5   0   0

```

4.5 a) 错误: `#include` 预处理程序指令后出现了分号。

改正:取消分号。

b) 错误:用赋值语句来为常量变量赋值。

改正:在 `const int arraySize` 声明中为常量变量赋值。

- c) 错误:在数组(`b[10]`)边界之外引用数组中的元素。

改正:把控制变量最后的值改为9。

- d) 错误:数组的下标出错。

改正:把语句改为 `a[ 1 ][ 1 ] = 5;`

## 练习题

### 4.6 填空题:

- C++ 把数值列表保存在\_\_\_\_\_内。
- 数组元素是通过\_\_\_\_\_发生关系的。
- 引用一个数组元素时,方括号内包含的位置编号称为\_\_\_\_\_。
- 数组 `p` 中的4个元素分别名为\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- 为数组命名,指定其类型及指定数组元素个数被称为\_\_\_\_\_数组。
- 把数组元素按升序或降序排列的过程叫做\_\_\_\_\_。
- 在双下标数组中,第一个下标(根据习惯)表示元素的\_\_\_\_\_,而第二个下标(根据习惯)则表示元素的\_\_\_\_\_。
- $m \times n$  的数组中包含\_\_\_\_\_行\_\_\_\_\_列和\_\_\_\_\_个元素。
- 数组 `d` 中第3行第5列的数组称为\_\_\_\_\_。

### 4.7 判断对错。如果错误,说明原因。

- 要引用数组内部特定的位置和元素,我们需要指定数组名称和特定元素的值。
- 数组声明会为数组保留存储空间。
- 为了表明为整数数组 `p` 保留100个位置,程序员编写了声明  
`p[ 100 ];`
- 要想把一个由15个元素组成的数组初始化为0,C++ 程序至少要包含一个 `for` 语句。
- 计算双下标数组元素总和的C++ 程序中,必须包含嵌套 `for` 语句。

### 4.8 针对以下任务,编写C++ 语句:

- 显示字符数组 `f` 中第7个元素的值。
- 在单下标浮点数组 `b` 的第4个元素中输入一个值。
- 把单下标整数数组 `g` 中的5个元素分别初始化为8。
- 合计并打印浮点数组 `c` 中的100个元素。
- 把数组 `a` 复制到数组 `b` 的第一部分。假设 `double a[ 11 ], b[ 34 ];`
- 针对由99个元素组成的浮点数组 `w`,判断并打印出这99个元素的最大值和最小值。

### 4.9 以 $2 \times 3$ 整数数组 `t` 为例。

- 为 `t` 编写声明。
- 数组 `t` 有多少行?
- 数组 `t` 有多少列?
- 数组 `t` 包含多少元素?
- 写出数组 `t` 第二行中所有元素的名称。

- f) 写出数组 `t` 第 3 列中所有元素的名称。
  - g) 编写一条语句,把 `t` 中第 1 行第 2 列中的元素设置为 0。
  - h) 编写一系列语句,把 `t` 的每个元素都初始化为 0。不要使用循环语句。
  - i) 编写一个嵌套 `for` 结构,把 `t` 的每个元素都初始化为 0。
  - j) 编写一条语句,通过终端输入 `t` 中各元素的值。
  - k) 编写一系列语句,判断并打印数组 `t` 中的最小元素。
  - l) 编写一条语句,显示 `t` 的第 1 行元素。
  - m) 编写一条语句,合计 `t` 的第 4 列中的元素。
  - n) 编写一系列语句,以整洁的表格形式打印数组 `t`。将列下标作为标题出现在首部,将行下标放在各行左侧。
- 4.10 利用单下标数组来解决以下问题。公司以底薪加提成的方式付给销售员工资。销售人员每周获得 200 美元的底薪,外加本周达到一定销售额的 9% 的提成。举个例子来说,如果一个销售人员一周的销售额是 5 000 美元,就会得到  $200 + 9\% \times 5\,000$  美元,也就是 650 美元的报酬。编写一个程序(利用一个计数器数组),判断有多少销售人员能获得以下范围内的报酬(假设每个销售人员的报酬都会取整):
- a) 200 ~ 299 美元
  - b) 300 ~ 399 美元
  - c) 400 ~ 499 美元
  - d) 500 ~ 599 美元
  - e) 600 ~ 699 美元
  - f) 700 ~ 799 美元
  - g) 800 ~ 899 美元
  - h) 900 ~ 999 美元
  - i) 1 000 美元以上
- 4.11 图 4.16 中的冒泡排序方法并不适用于大型数组。执行以下简单修改提高冒泡排序方法的性能:
- a) 第一遍排序后,大型数字总会成为数组中编号最大的元素;第二遍排序后,就有两个编号最大的数字,依此类推。这样一来,就不是每次进行 9 次比较,而是第二遍比较 8 次,第 3 遍比较 7 次,依此类推。
  - b) 数组中的数据可能已经按正确或接近于正确的顺序进行排列,所以如果比较次数较少也能达到理想效果时,为什么一定要比较 9 次呢? 修改排序方法,查看每次比较后是否数据有无交换,如果没有,那么数据肯定已经采用了正确的排序。如果有交换,则说明至少还需要一次排序。
- 4.12 编写一条语句,执行以下单下标数组操作:
- a) 把整数数组 `counts` 的 10 个元素初始化为 0。
  - b) 把整数数组 `bonus` 的 15 个元素分别加 1。
  - c) 通过键盘输入 `double` 数组 `monthlyTemperatures` 的 12 个值。
  - d) 以列的方式打印整数数组 `bestScores` 的 5 个值。

4.13 指出下列各条语句中的错误:

a) 假设:

```
char str[ 5 ];  
cin >> str;    //User types hello
```

b) 假设:

```
int a[ 3 ];  
cout << a[1] << " " << a[ 2 ] << " " << a[ 3 ] << endl;
```

c) double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.000 };

d) 假设:

```
double d[ 2 ][ 10 ];  
d[ 1, 9 ] = 2.345;
```

4.14 修改图 4.17 中的程序,令函数 mode 能处理模值的连接。与此同时,还要修改函数 median,令两个中间元素是偶数个元素数组中的平均值。

4.15 利用单下标数组解决以下问题。读取 20 个数,各数的取值在 1~100 之间。读取每个数时,只打印不与已读过数相同的数。假设“最糟糕的情形”是这 20 个数都不相同。请用最小的数组来解决此问题。

4.16 标出  $3 \times 5$  双下标数组 sales 中的元素,通过程序段

```
for ( row = 1; row < 3; row ++ )  
    for ( column = 0; column < 5; column ++ )  
        sales[ row ][ column ] = 0;
```

指出各元素被设置为 0 的顺序。

4.17 编写一个程序,模拟掷两个骰子。程序将用 rand 来投掷第一个骰子,再次用 rand 函数来投掷第二个骰子,然后计算这两个值的和。注意:由于每个骰子会显示从 1 到 6 之间的值,所以掷两个骰子的值之和为 2 到 12,其中,7 最为常见,2 和 12 出现的机会则较少。图 4.28 展示了掷两个骰子所得值的 36 种组合。程序要投掷这两个骰子的次数为 36 000 次。用单下标数组估算每次可能出现的两数之和。用表格的形式打印结果。此外,还要判断两数之和是否合理(例如,如果两数之和为 7 的方式有 6 种,那么所有投掷次数中,和为 7 的概率将是六分之一)。

|   | 1 | 2 | 3 | 4  | 5  | 6  |
|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

图 4.28 掷两个骰子时,所得结果值之和的 36 种组合

## 4.18 指出以下程序的用途。

```

1  //ex04_18.cpp
2  #include <iostream>
3
4  using std::cout;
5  using std::endl;
6
7  int whatIsThis( int [], int );
8
9  int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14     int result = whatIsThis( a, arraySize );
15
16     cout << "Result is " << result << endl;
17     return 0;
18 }
19
20 int whatIsThis( int b[], int size )
21 {
22     if ( size == 1 )
23         return b[ 0 ];
24     else
25         return b[ size - 1 ] + whatIsThis( b, size - 1 );
26 }

```

## 4.19 编写一个程序,运行 1 000 次掷骰子游戏,并回答以下问题:

- a) 第一次、第二次……第 20 次以及第 20 次以后的游戏中,共赢了多少次?
- b) 第一次、第二次……第 20 次以及第 20 次以后的游戏中,共输了多少次?
- c) 掷骰子游戏中,赢的概率为多少(注意:你会发现掷骰子游戏是最公平的赌博游戏。请解释为什么?)。
- d) 掷骰子游戏的平均时间为多长?
- e) 玩游戏的时间越长,是否意味着赢的机会越多?

## 4.20 (航班订票系统)一家小型的航空公司新近购买了一台计算机,打算将其用于自动订票系统。要求你为这套新系统编写程序。编写一个程序,为该公司惟一一架飞机的每次航班分配座位(该飞机的载客量为 10 人)。

程序将显示以下菜单选项:Please type 1 for "First Class" 和 Please type 2 for "Economy"。如果客人输入 1,程序就应该为其分配头等舱的座位(1~5 号)。如果客人输入 2,程序就应该为其分配经济舱的座位(6~10 号)。程序要打印标有客人座位号的登机牌,并、以及该座位位于头等舱还是经济舱。

用单标数组表示飞机座位图。将该数组中的所有元素初始化为 0,表明所有座位都是空的。分配每个座位时,就要把数组中相应元素设置为 1,表明该座位已订。

当然,程序不能把已经订过的座位重新分配给别人。头等舱满座时,程序应询问客人

是否愿意接受非吸烟区的座位(反之亦然)。如果客人愿意,就为其分配相应的座位。如果不愿意,就打印消息“Next flight leaves in 3 hours”(下一次航班将于3小时后离港)。

#### 4.21 指出下列程序的用途。

```

1  //ex04_21.cpp
2  #include <iostream>
3
4  using std::cout;
5  using std::endl;
6
7  void someFunction( int [], int );
8
9  int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] =
13         32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14
15     cout << "The values in the array are: " << endl;
16     someFunction( a, arraySize );
17     cout << endl;
18     return 0;
19 }
20
21 void someFunction( int b[], int size )
22 {
23     if ( size > 0 ) {
24         someFunction( &b[ 1 ], size - 1 );
25         cout << b[ 0 ] << " ";
26     }
27 }
```

#### 4.22 用双下标数组解决以下问题。假设一家公司有4个销售员(1~4),他们负责销售5种不同的商品(1~5)。一天,每个销售员都要报告每种商品的销售情况。每份报告中要包含以下内容:

- a) 销售员编号;
- b) 商品编号;
- c) 当天自己所负责商品的销售总额。

这样一来,每个销售员每天都要上交0~5份销售报告。假设上月每份报告中的信息都是可用的。编写一个程序,读取这几类信息,了解上个月的销售情况并对每个销售员和每中商品进行汇总。所有汇总将保存在双下标数组 sales 内。处理完上个月的所有信息之后,以表格(列代表销售员,行代表商品)形式打印结果。通过横向交叉求和,获得上个月每种商品的销售总额;通过竖向交叉求和,获得上个月每个销售员的销售总额。打印输出中应该包含右边的行小计和底部的列小计。

#### 4.23 (海龟图) Logo 语言是个人计算机用户最熟悉语言之一,正是它,使“海龟图”的概念



广为人知。假设有这样一个机械海龟,它在C++ 程序的控制下,在房间中移动。海龟在两个方向中选择一个方向(向上或向下)握住画笔。画笔向下,海龟就会随着画笔的移动,跟踪其移动的形状;当画笔向上,海龟就自由移动,不会留下任何移动的轨迹。在这个问题中,要求你模拟海龟的操作,并创建一个计算机化的画板。

用一个  $20 \times 20$  的数组 floor(该数组已被初始化为 0)。从其中已包含命令的数组中读取命令。随时跟踪海龟的当前位置,以及画笔当前所处的状态(向上或向下)。假设海龟始终从在 0 这个位置,那么画笔就会从 0 开始向上。程序必须处理的海龟命令如下:

| 命令   | 描述                   |
|------|----------------------|
| 1    | 画笔向上                 |
| 2    | 画笔向下                 |
| 2    | 右转                   |
| 4    | 左转                   |
| 5,10 | 向前移动 10 格(或少于 10 格)  |
| 6    | 打印 $20 \times 20$ 数组 |
| 9    | 数据结束(标记)             |

假设海龟处于靠近平面中心的某处。以下程序

```

2
5,12
3
5,12
3
5,12
1
6
9

```

将绘制并打印一个  $12 \times 12$  的正方形,并在结束处保持画笔向上。当画笔向下,海龟随之而移动时,把 floor 数组中的相应变量设置为 1。给出命令 6(打印)时,只要数组中有 1,就要显示一个星号(\*)或自选的其他符号。只要数组中有 0,就显示空白。编写一个程序,实现这里讨论的海龟图功能。编写几个海龟图程序,绘制其他更有趣的图形。添加别的命令增减海龟图语言的功能。

- 4.24 (骑士旅行)对于国际象棋,最有趣的问题之一是骑士旅行问题。这个问题最早是由瑞士数学家及自然科学家欧拉提出来的。问题如下:名为骑士的棋子能否在空棋盘上走完在 64 个棋盘格,每格只能只走一次?下面将详细讨论这个有趣的问题。

在国际象棋中,骑士按 L 路线移动(一个方向两格,然后再往垂直方向移动一格,其行走路线相当于中国象棋中的马)。因此,从空棋盘中央的方格算起,骑士有 8 种移动方式(从 0 到 7),如图 4.29 所示。

- a) 在一张纸上画出  $8 \times 8$  的棋盘,试着手工移动骑士。在移动的第一个方格中标上 1,第二个方格中标上 2,第 3 个方格中标上 3,依此类推。开始移动骑士之前,估计自

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   |   |   |   |   |   |
| 1 |   |   |   | 2 |   | 1 |   |   |
| 2 |   |   | 3 |   |   |   | 0 |   |
| 3 |   |   |   |   | k |   |   |   |
| 4 |   |   | 4 |   |   |   | 7 |   |
| 5 |   |   |   | 5 |   | 6 |   |   |
| 6 |   |   |   |   |   |   |   |   |
| 7 |   |   |   |   |   |   |   |   |

图 4.29 骑士的 8 种旅行路线

己要走多远,并记住一共要走 64 步。估计以下自己要走多远,而且当前走的这一步是否接近自己的预期?

- b) 接下来开发一个程序,在棋盘上移动骑士。棋盘用  $8 \times 8$  的双下标数组 `board` 来表示,每个方格都被初始化为 0。我们用水平和垂直组件来描述 8 种可能的行走路线。例如图 4.25 中,0 类型的移动是这样的:沿水平方向右移 2 格,沿垂直方向上移一格。2 类型的移动则是:水平方向左移 1 格,垂直方向上移 2 格。水平方向的左移和垂直方向的上移均用负数来表示。这 8 种移动方式可用 2 个单下标数组 `horizontal` 和 `vertical` 来表示,如下所示:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

变量 `currentRow` 和 `currentColumn` 表明骑士当前位置的行与列。要执行 `moveNumber` 类型的移动(也就是 `moveNumber` 在 0~7 之间),程序应该采用语句

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

令计数器在 1 到 64 之间变化,记录骑士在每一格中移动的最新计数。记得测试每个可能的移动,以确定骑士是否已访问过该方格,当然,还要测试每个可能的移动,确定骑士不会跳出棋盘。现在,编写一个在棋盘上移动骑士的程序。然后运行它,看骑士要走完 64 个方格,应走多少步?

- c) 尝试编写并运行骑士旅行的程序后,你可能已经发现了部分有价值的东西。我们将利用它们来开发一个用于推测骑士移动路线的推断性(或策略性)程序。推断不能保证成功,但经过周密设计的推断能有效提高成功的机会。如果仔细观察,你会发现外层的方格比棋盘中间的方格更麻烦。事实上,最麻烦的,而且最难到达的是位于棋盘四角的方格。

你可能会凭直觉,把骑士先移到最难到达的方格,在骑士走完所有旅程时,达到最容易到达的方格。这样成功的机率会比较高。

我们将根据每个方格的到达难度,进行分类,并且始终令骑士位于最难以达到的那个方格(当然骑士还是得按 L 路线行走),通过这一方式,开发一个“访问难度推断程序”。我们在双下标数组 `accessibility` 中标上每个方格能访问的格数。在空棋盘上,位于中部的每个方格的访问难度值为 8,位于四角的每个方格的访问难度值为 2,其他所有方格的访问难度分别为 3,4 和 6,如下所示:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 3 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

现在,用访问难度推断程序编写一个骑士旅程序。任何时候,骑士都应该移到访问难度最高的那个方格。对于相邻的方格,骑士可能会移到相邻方格中的任何一个中。因此,骑士的旅行应该从位于四角的方格开始(注意,随着骑士在棋盘中的移动,程序要能随着访问方格个数的增多,而逐渐减少方格的访问难度。通过这种方式,在旅行中的任何时刻,每个方格的访问难度都应该是访问这个方格应走的步数)。然后运行程序。看骑士能否走完所有旅程? 现在修改程序,从棋盘上的任何一个方格开始遍历 64 个方格的旅行。看自己是否能访问多少个方格?

- d) 编写一个骑士旅程序,在遇到两个或多个方格相邻时,该程序能通过从相邻方格能访问到的方格选择下一个方格。程序应把骑士移到下一个访问难度较高的方格。

4.25 (骑士旅行:强制执行方式)在练习 4.24 中,我们开发了骑士旅行问题的解决方案。其中所用的方法“访问难度推断法”生成了许多解决方案和并能有效地实行。

随着计算机技术的突飞猛进,我们可以凭借着其强大的计算能力,利用相对简单的算法来解决更多的问题。我们把这种方式称为“强制执行”。

- a) 用随机数生成程序让骑士在棋盘上随意移动(当然,走 L 路线)。程序应走完所有方格。骑士能走多远?
- b) 多数情况下,前面的程序会生成一个比较短的旅程。现在修改这个程序,执行 1 000 次旅行。用单下标数组跟踪每次旅行所走的步数。程序执行了 1 000 次旅行后,应该以表格形式打印出这些信息。最佳结果是什么?
- c) 多数情况下,前面的程序会给出部分较好的旅程,而不是完整的旅程。现在,“去掉次数限制”,让程序一直运行,直到它走完棋盘上所有方格为止(注意,在计算能力强大的计算机上运行这个修改后的程序,可能要花数小时的时间)。再次提醒大家注意,以表格的形式保留每次旅行所走的方格数,并在首次走完整个棋盘时打印该表。程序要尝试多少次才能找到首次走完棋盘的方式? 以及所花的时间是多少?
- d) 针对骑士旅行问题,比较其强制实行方式和访问难度推断方式。哪种方式更需要我们深入研究问题? 哪种算法更难以实现? 哪种方式对计算能力要求更高? 我们是否确定可以通过访问难度推断方式来获得走完棋盘所有方格的方法? 我们是否能确定可以通过强制实行方式来走完棋盘上所有方格? 权衡强制实行方式的利与弊。
- 4.26 (八皇后问题)关于国际象棋,另一个有趣的问题是八皇后问题。简单地说就是空棋盘上是否能防止八个皇后,要保证这八个皇后之间不会互相“攻击”(也就是面对面),也就是说同一行、同一列和同一个对角线上不能出现 2 个皇后? 用练习 4.24 的设计思路来阐明解决八皇后问题的推断法。运行程序(提示:可以为棋盘上的每个方格指定一个值,表明一个皇后位于一个方格时,空棋盘上有多少方格可以排除。棋盘各角的值都指定为 22,如图 4.26 所示)。在 64 个方格中放入“要排除”的数之后,就会得出相应的推断结果:把下一个皇后放在排除数最少的方格中,为什么会凭直觉选用这种方式?



图 4.30 棋盘左上角中放入一个皇后以后,可排除 22 个方格

- 4.27 (八皇后问题:强制实行)这个练习题,要求你开发几个强制实行方式,解决练习题 4.26 中的八皇后问题。
- a) 利用练习题 4.26 中介绍的随机强制实行方式来解决八皇后问题。
- b) 利用穷举法,也就是尝试 8 个皇后在棋盘上的所有组合。
- c) 为什么穷举强制实行法不宜用于解决骑士旅行问题?
- d) 比较并对照随机强制实行法和穷举强制实行法。
- 4.28 (骑士旅行问题:封闭旅程测试)在骑士旅行中,所谓完整的旅程指的是骑士要一一经

历棋盘中的 64 个方格,且一个方格只能经历一次。闭合旅程指的是骑士最后又回到起点。修改练习题 4.24 中编写的骑士旅程序,测试闭合旅程是否会经历棋盘上的每个方格。

4.29 (Eratosthenes 筛选法)质数是只能被 1 及其本身整除的数。Eratosthenes 筛选法是一种寻找质数的方法。它的方式如下:

- a) 创建一个数组,将它的所有元素初始化为 1(真)。下标为质数的数组元素将保持 1 不变,其他数组元素最后设为 0。
- b) 从数组下标 2(下标 1 必定是质数),每次发现其值为 1 的数组元素,就会在数组的其他元素中循环,并将下标为该下标倍数的那个元素设为 0。对于数组下标 2,数组中下标为 2 的倍数的所有元素都会被设为 0(下标 4,6,8,10 等);对于数组下标 3,数组中下标为 3 的倍数的所有元素都会被设为 0(下标 6,9,12,15 等等);以此类推。

完成这个过程之后,如果数组元素仍然为 1,就表明其下标是质数。这些下标随后会打印出来。编写一个程序,令其使用一个由 1 000 个元素组成的数组,判断并打印 1 到 99 之间的所有质数。忽略数组中的元素 0。

4.30 (桶的排序)关于桶的排序,要从排序单下标正整数数组和双下标整数数组(行下标在 0~9 之间,列下标在 0~ $n-1$  之间,这里  $n$  指的是要排序的那个数组中的数值个数)开始,双下标数组中的每一行都被称为一个桶。编写一个函数 bucketSort,令其采用整数数组及其长度作为参数,并执行以下操作:

- a) 对于单下标数组中的每个值,根据其数位,将它们放入桶数组的列中。例如,97 放入第 7 行,3 放入第 3 行,100 放在第 0 行。这就是“分布传递”。
- b) 一行一行地在桶数组中循环,并把数值复制到原始数组。这就是“收集传递”。上述数值在单下标数组中的新顺序将是 100,3 和 97。
- c) 针对随后的每个数位(十分位、百分位、千分位等),重复上述排序过程。

在第二遍排序过程中,100 被放入第 0 行,3 被放入第 0 行(因为 3 没有十分位)而 97 位于第 9 行。收集传递完成之后,单下标数组中数值的顺序则是 100,3 和 97。第三次收集之后,100 被放入第 1 行,3 被放入第 0 行,97 也被放入第 0 行。最后一次收集传递之后,技能获得原始数组的排序结果。

注意,双下标桶数组的长度是要排序的整数数组长度的 10 倍。和冒泡排序相比,这种排序方式具有更好的性能,但对内存容量有较高的要求。冒泡排序只需要额外的一个数据元素的空间。这里有一个空间交换时间的范例:桶排序占用的内存空间比冒泡排序多,但具有性能上的优势。这种方式的桶排序在每次收集传递时,需要把所有数据复制到原始数组。另一种方式是创建第二个双下标桶数组,并在这两个桶数组间重复交换数据。

## 递归练习

4.31 (选择排序)选择排序用于查找数组中的最小元素。然后,把最小的元素与数组中的第一个元素进行交换。接着从数组中第二个元素的子数组开始重复这个过程。每次排

序都会把一个元素放到合适的位置。这种排序方式类似于冒泡排序:对于  $n$  个元素组成的数组,必须执行  $n-1$  次排序,对于每个子数组,还必须进行  $n-1$  次比较,找出最小数值。处理过的子数组中只包含一个元素时,就表明数组排序已经完成。编写一个递归函数 `selectionSort`,执行这个算法。

- 4.32 (回文)回文即一种字符串,无论正读,还是反读,都会拼出同样的单词或短语。比如“radar”和“able was I ere is saw elba”以及“a man a plan a canal panama”(如果忽略空格的话)。编写一个递归函数 `testPalindrome`,令其在数组中包含回文字符串时返回 `true`,反之则返回 `false`。函数应当忽略字符串中的空格。
- 4.33 (线性搜索)修改图 4.19 中的程序,用递归函数 `linearSearch` 对数组执行线性搜索。函数应当接收一个整数数组及该数组的长度作为参数。如果找到搜索键,就返回数组下标;反之则返回 `-1`。
- 4.34 (二元搜索)修改图 4.20 中的程序,用递归函数 `binary - Search` 来执行对数组的二元搜索。函数应接收一个整数数组、起始下标和结束下标作为参数。如果找到搜索键,就返回数组下标;反之则返回 `-1`。
- 4.35 (八皇后问题)修改练习题 4.26 中创建的程序,用递归的方式解决这一问题。
- 4.36 (打印数组)编写一个递归函数 `printArray`,将一个数组及其长度作为参数,并不返回任何数值。函数应当在收到长度为 0 的数组时停止处理并返回。
- 4.37 (逆向打印字符串)编写一个递归函数 `stringReverse`,将其中包含字符串的字符数组作为参数,逆向打印字符串,并不返回任何数值。函数应当在碰到空字符串时停止处理并返回。
- 4.38 (找出数组中的最小值)编写一个递归函数 `recursiveMinimum`,将一个整数数组及其长度作为参数,并返回数组中的最小值。函数应当在收到长度为 1 个元素的数组时停止处理并返回。

# 第 5 章 指针和字符串

## 学习目标

- 能够使用指针
- 能够使用指针引用调用将函数传送至函数
- 理解指针、数组和字符串之间的紧密联系
- 理解指针指向函数的用法
- 能够声明并且使用由字符串组成的数组

## 5.1 简介

本章将介绍C ++编程语言中最强大的特性:指针。指针也是C ++语言中最难掌握的概念之一。第3章曾介绍引用可用于引用调用。指针使程序能够模拟按引用调用,还可以生成和操作动态数据结构,即可以伸缩的数据结构,例如:链表、队列、堆栈和树。本章在解释指针基本概念的同时,强调了数组、指针和字符串之间的密切关系,此外包括了一组很好的字符串处理练习题。

第6章介绍指针结构的用法。第9章和第10章将介绍通过指针和引用执行面向对象编程。第15章会介绍动态内存管理技术并提供创建及使用动态数据结构的例子。

指针数组和指针字符串从C语言派生而来。稍后我们将把数组和字符串视为成熟对象进行描述。

## 5.2 指针变量声明和初始化

指针变量存储值为地址。通常,变量包含一个特定值,而指针包含的是这个特定值所在的地址。从这个意义上讲,变量名直接引用变量值,而指针是间接引用变量值(如图5.1所示)。通过指针引用变量值称为间接引用。

和其他变量一样,指针必须先声明后使用。例如声明

```
int *countPtr, count;
```

变量countPtr的类型是\*int(即指向整型值的指针),或者叫做“countPtr是指向int的指针”,或者“countPtr指向一个整数类型对象”。同时变量count被声明为整数,而不是指向整型值的指针。符号“\*”在指针声明里必须位于变量前。每一个指针变量被声明时必须在前面加星号(\*)。例如声明

```
double *xPtr, *yPtr;
```

xPtr和yPtr都是指向复数型的指针。当“\*”用于声明中,它表示这个变量是指针变量,指针可指向任何数据类型的对象。

**常见编程错误 5.1** 假设用于声明指针的星号(\*)可以分配给声明中用逗号分隔的指针变量列表中的所有指针变量名,就能将指针声明为非指针。每个指针在声明时必须在名称前面加星号(\*)。

**良好编程习惯 5.1** 尽管并不一定要这样做,但是变量名中包含字母Ptr能够更清楚地表示这个变量是指针变量并且需要适当处理。

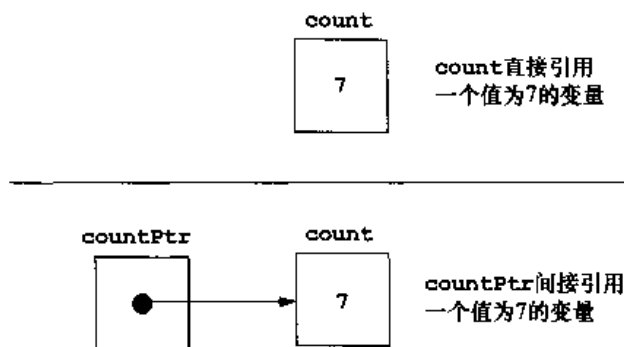


图 5.1 直接和间接引用变量

指针应该在声明时或者在赋值语句中初始化。指针可以被初始化为 0、NULL 或地址,一个数值为 0 或 NULL 的指针不指向任何内容。符号化常量 NULL 在头文件 `<unistd.h>` (和另外几个标准库头文件)中定义。指针初始化为 NULL 和初始化为 0 是相同的,但是在 C++ 中,0 是优先选用的。赋值后的 0 会转换为一个相应类型的指针。数值 0 是整数值,只有它才可不经过整数到指针的类型转换过程就可以直接赋值为指针变量。5.3 节将描述如何把变量地址赋给指针。

**测试和调试提示 5.1** 指针初始化是为了防止指向未知的和未经初始化的内存区域。

### 5.3 指针操作符

&、地址、操作符是一元操作符,它可以返回其操作数地址。如声明

```
int y = 5;
int *yPtr;
```

如果语句

```
yPtr = &y;
```

将变量 y 的地址赋给指针变量 yPtr,然后将变量 yPtr 读为“指向”y。图 5.2 展示了上述赋值语句被执行后的内存示意图。在此图中,我们通过从指针到它所指向的对象之间画一个箭头表示了“指向关系”。

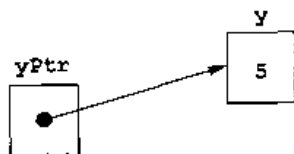


图 5.2 指针指向内存中整数变量的示意图



图 5.3 展示了内存中的指针表达式。假设整型变量 `y` 被存放在地址 600000, 指针变量 `yPtr` 被存放在地址 500000。地址操作符的操作数必须是左值(即该值能够被赋值, 如变量名); 地址操作符不能用于常量前, 也不能用于引用结果表达式, 或用存储类 `register` 声明的变量前。



图 5.3 表示内存中的 `y` 和 `yPtr`

“`*` 操作符”通常称为“间接操作符”或“复引用操作符”, 它能够返回操作数(即指针)指向对象的同义词、别名或浑名。例如(再次引用图 5.2), 语句

```
cout << *yPtr << endl;
```

指向变量 `y` 的值, 即 5, 其方式如同语句

```
cout << y << endl;
```

以这种方式使用 `*` 被称为复引用指针。注意, 复引用指针也可以用在赋值语句左边, 如语句

```
*yPtr = 9
```

将数值 9 赋给 `y`, 如图 5.3 所示。复引用指针也可以用于接收输入值, 例如

```
cin >> *yPtr
```

复引用指针是 lvalue, 即“左值”。

**常见编程错误 5.2** 如果指针没有初始化或没有指定指向内存中的特定地址, 而复引用指针可能造成致命的运行时错误, 或者意外修改重要数据, 虽然程序仍可运行, 但得到的结果会是错的。

**常见编程错误 5.3** 复引用指针是语法错误。

**常见编程错误 5.4** 复引用 0 指针通常是一个致命的运行时错误。

图 5.4 中的程序演示了指针操作符, 在该示例中, 利用 `<<` 以十六进制整数形式输出内存地址(有关十六进制整数的详情, 参见附录 C)。

```

1 //Fig. 5.4; fig05_04.cpp
2 //Using the & and * operators
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int a;          //a is an integer
11     int *aPtr;      //aPtr is a pointer to an integer
12
13     a = 7;
14     aPtr = &a;      //aPtr set to address of a
  
```

```
15
16     cout << "The address of a is " << &a
17         << " \nThe value of aPtr is " << aPtr;
18
19     cout << " \n\nThe value of a is " << a
20         << " \nThe value of *aPtr is " << *aPtr;
21
22     cout << " \n\nShowing that * and & are inverses of "
23         << "each other.\n&*aPtr = " << &*aPtr
24         << " \n*&aPtr = " << *&aPtr << endl;
25     return 0;
26 }
```

输出结果:

```
The address of a is 006AFDF4
The value of aPtr is 006AFDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 006AFDF4
*&aPtr = 006AFDF4
```

图 5.4 & 与 \* 指针操作符

**可移植性提示 5.1** 指针输出形式与机器有关。有的系统用十六进制整数,有的系统则用十进制整数。

注意,地址 `a` 和变量值 `aPtr` 在输出中是完全相同的,说明地址 `a` 确实赋给了指针变量 `aPtr`。`&` 和 `*` 操作符是相反的,两者作用于 `aPtr`,会打印出相同的结果。图 5.5 显示了如前所述的操作符的优先级和结合性。

| 优先级                                       | 结合性  | 类型     |
|-------------------------------------------|------|--------|
| <code>() {}</code>                        | 从左到右 | 高      |
| <code>++, -- static_cast&lt;类型&gt;</code> | 从左到右 | 置后     |
| <code>++, --, +, -, !, &amp;, *</code>    | 从右到左 | 一元     |
| <code>*, /, %</code>                      | 从左到右 | 乘      |
| <code>+, -</code>                         | 从左到右 | 加      |
| <code>&lt;&lt;, &gt;&gt;</code>           | 从左到右 | 插入/读取  |
| <code>&lt;, &lt;=, &gt;, &gt;=</code>     | 从左到右 | 关系     |
| <code>==, !=</code>                       | 从左到右 | 相等     |
| <code>&amp;&amp;</code>                   | 从左到右 | 逻辑 AND |
| <code>  </code>                           | 从左到右 | 逻辑 OR  |
| <code>?:</code>                           | 从左到右 | 条件     |
| <code>=, +=, -=, *=, /=, %=</code>        | 从右到左 | 赋值     |
| <code>,</code>                            | 从左到右 | 逗号     |

图 5.5 操作符的优先级和结合性

## 5.4 按引用调用函数

在C++中,有3种方法可以将参数传递到函数:传值调用、使用引用参数按引用调用和用指针参数(按引用调用)。第3章,利用引用参数比较了传值调用和引用调用。本章将集中介绍用指针参数按引用调用。

如第3章所述,return可用于把数值从被调函数返回主调函数(或者不返回变量值就可以从被调函数返回控制)。后文将介绍用引用参数将参数传递到函数,使函数修改原始的参数值(因此,可以从函数“返回”多个值),或者将大型数据对象传递到函数,并且避免了传值调用传递对象的开销(即复制对象需要的开销)。类似于引用,指针也可用于修改调用主调函数中一个或多个变量,或者将大型数据对象指针传递给函数而避免了引用调用传递对象的开销。

在C++语言中,程序员可以用指针和间接操作符来模拟引用调用(更确切地讲,引用调用是在C语言编程中实现的)。若调用函数要修改参数,传递的是参数的地址。常用方法是在数值将被修改的变量名前加地址操作符&。

正如第4章中所述,数组不能用操作符&来传递,因为数组是数组中的开始位置。数组名等同于arrayName[0](即一个数组名就是一个指针)。当变量地址传递到函数,间接操作符\*可以在函数中用来形成变量名称的同义词、别名或浑名,这将会在主调函数内存位置上依次修改数值(前提是变量没有被声明const)。

图5.6和图5.7中的程序是计算整数立方函数的两个版本:cubeByValue和cubeByReference。如图5.6所示,使用传值调用将变量number传递给函数cubeByValue。函数cubeByValue求出参数的立方并且用return语句将新值返回到main,在main中,将该新值赋给number。在修改变量值之前,可以检查函数调用的结果。例如:在这个程序中,我们本可以把cubeByValue的结果存放在另一个变量里,在检查了数值的合理性之后,再检查其值并将结果赋给number。

```
1 //Fig. 5.6: fig05_06.cpp
2 //Cube a variable using call-by-value
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int cubeByValue( int ); //prototype
9
10 int main()
11 |
12 |     int number = 5;
13 |
14 |     cout << "The original value of number is " << number;
15 |     number = cubeByValue( number );
16 |     cout << " \nThe new value of number is " << number << endl;
17 |     return 0;
18 |
```

```

19
20 int cubeByValue( int n )
21 |
22     return n * n * n;    //cube local variable n
23 |

```

输出结果:

The originl value of number is 5

The new value of number is 125

图 5.6 使用传值调用求出参数的立方

```

1 //Fig.5.7: fig05_07.cpp
2 //Cube a variable using call-by-reference
3 //with a pointer argument
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 void cubeByReference( int * );    //prototype
10
11 int main()
12 {
13     int number = 5;
14
15     cout << "The original value of number is " << number;
16     cubeByReference( &number );
17     cout << "\nThe new value of number is " << number << endl;
18     return 0;
19 }
20
21 void cubeByReference( int *nPtr )
22 {
23     *nPtr = *nPtr * *nPtr * *nPtr;    //cube number in main
24 }

```

输出结果:

The original value of number is 5

The new value of number is 125

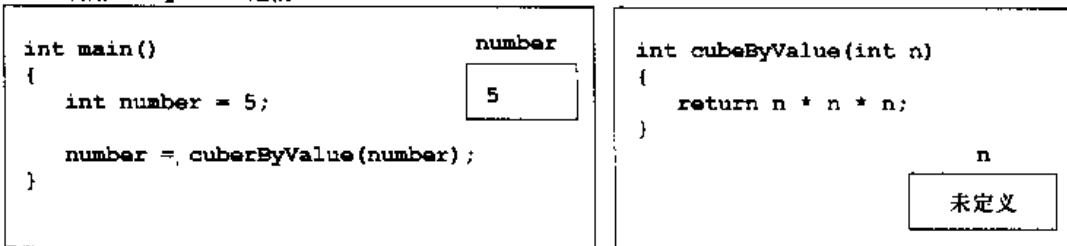
图 5.7 用指针参数按引用调用求出参数的立方

图 5.7 的程序用引用调用传递变量 `number` (传递 `number` 的地址)到函数 `cubeByReference`。函数 `cubeByReference` 取 `nPtr` (`int` 指针)作为参数。函数复引用指针求出 `nPtr` 所指值的立方,并改变了 `main` 中的 `number` 值。图 5.8 和图 5.9 以图表的方式分别分析了图 5.6 和图 5.7 中的程序。

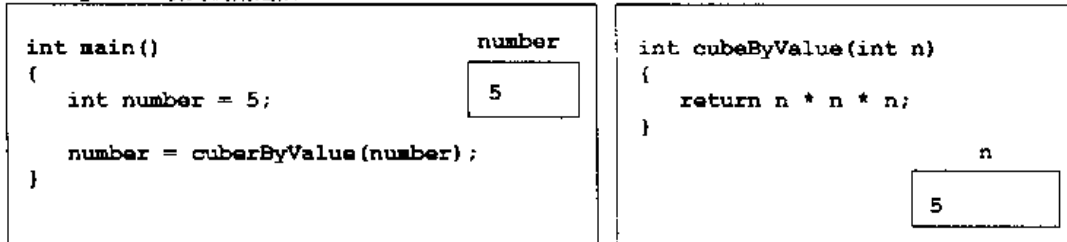
**常见编程错误 5.5** 当需要获得指针指向的变量值时,不使用复引用指针会出错。

接收地址参数的函数必须定义成接收地址的指针参数。例如:函数 `cubeByReference` 首部定义如下

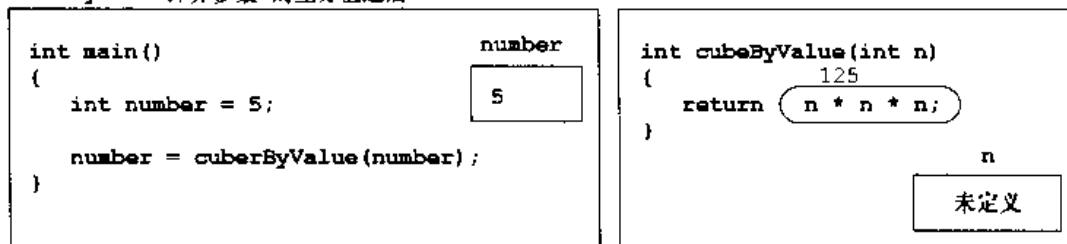
main调用cubeByValue之前:



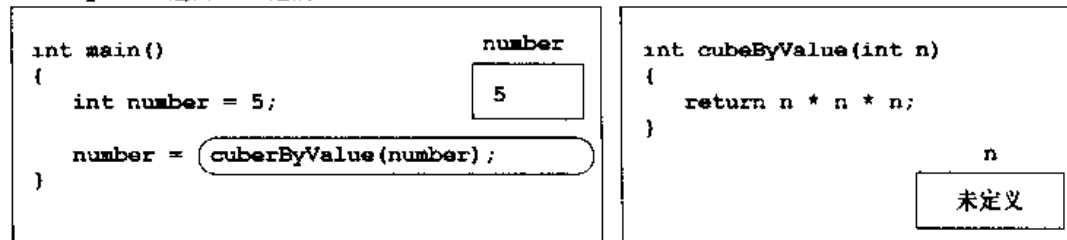
cubeByValue收到调用之后:



cubeByValue计算参数n的立方值之后:



cubeByValue返回main之后:



main完成对number的赋值后:

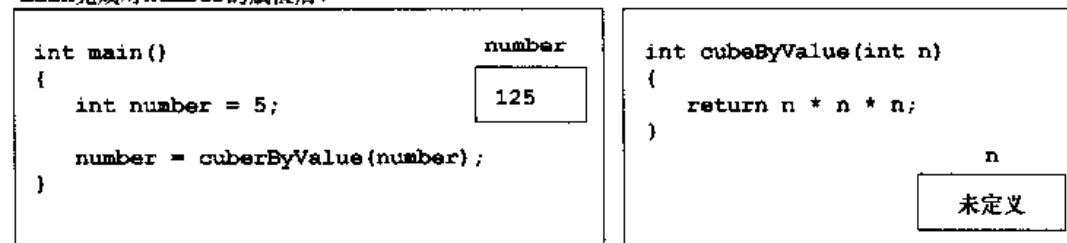


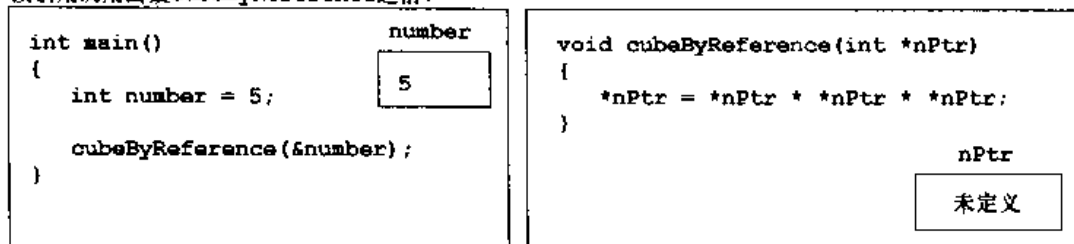
图 5.8 典型传值调用的分析

```
void cubeByReference(int *Ptr)
```

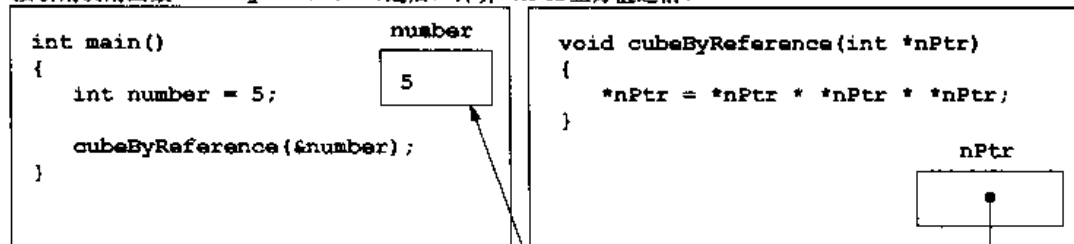
这个函数首部指定函数 cubeByReference 把整型变量(即整型指针)地址作为参数接收,把这个地址存放在 Ptr 而不返回值。

CubeByReference 的函数原型包含括号中的 int \*,和其他变量类型一样,不需要在函数

按引用调用函数 `cubeByReference` 之前:



按引用调用函数 `cubeByReference` 之后, 计算 `*nPtr` 立方值之前:



计算 `*nPtr` 立方值之后:

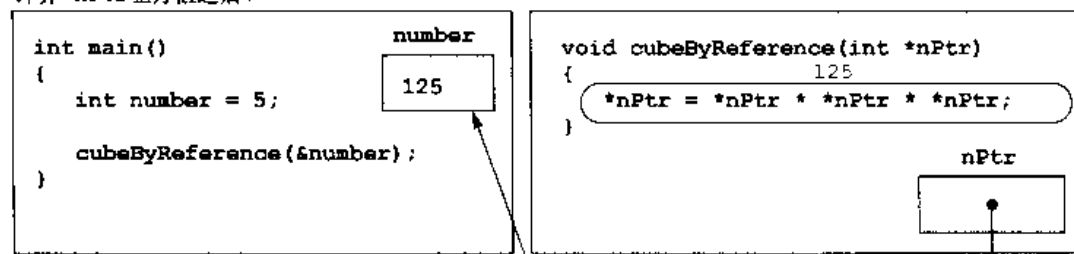


图 5.9 典型的用指针参数按引用调用分析

原型中包括指针名。参数名仅用于程序中的说明,编译器将其忽略。

在需要单下标数组参数的函数头部和函数原型中,可以用 `cubeByReference` 参数表中的指针符号。编译器并不区别接收指针的函数和接收单下标数组的函数。当然,这意味着函数必须“知道”什么时候接收数组或者进行引用调用的单个变量。当遇到 `int b[]` 形式的单下标数组的函数参数时,编译器将参数转换成指针符号 `int * const b` (读成 `b` 是指向整数的常量指针,有关常量的知识请参见 5.5 节)。声明函数参数为单下标数组的两种形式可以交换使用。

**良好编程习惯 5.2** 如果主调函数没有明确要求被调函数在主调函数环境中修改参数变量值,使用传值调用将参数传递到函数,这是最低权限原则的另一个例子。

## 5.5 使用带指针的 `const` 限定符

程序员可通过 `const` 限定符通知编译程序不能修改特定变量值。

**软件工程知识 5.1** `const` 限定符可以执行最低权限原则。利用最低权限原则正确设计软件既可显著减少调试时间和负面影响,又可简化程序的修改和维护。

**可移植性提示 5.2** 尽管 ANSI C 语言和 C++ 已经较好地定义了 `const` 限定符,但有些编译器仍无法正确实现。

过去几年,大量 C 语言遗留代码都没有使用 `const` 限定符,所以使用旧版 C 语言代码的软件工程具有较大的改进余地。目前也有很多使用 ANSI C 语言和 C++ 语言的程序员没有在自己的程序中使用 `const` 限定符,因为他们接触编程时使用的就是 C 语言较早的版本。这些编程人员因此错失了很多改进软件工程的好机会。

函数参数使用(或不使用)`const` 有 6 种可能性,其中两种是通过传值调用传递参数,另外 4 种是按引用调用传递参数。如何在这 6 种可能性中选择呢?最低权限原则给我们提供了很好的解决方法。在参数中向函数提供完成指定任务所需的数据访问,但不要提供更多权限。

第 3 章曾经介绍了使用传值调用传递参数,在函数调用中生成了参数副本(或者参数),并被传递到函数。如果副本在函数中被修改,主调函数的原值不会有任何改变。在很多案例中,一个传递到函数的值被修改,函数也就完成了它的任务。但是在有些情况下,尽管被调函数只是操作原值的副本,被调函数中的值也不应该被修改。

大家应该考虑函数读取一个单下标数组,及其长度为参数,并打印这个数组。函数应对数组进行循环,并且单独输出每一个数组元素。函数体中用数组长度来确定数组的最高下标,因此当打印完成后,循环即中止。在函数体中,数组的长度不改变。

**软件工程知识 5.2** 如果传递到函数体中的值不变(或不会改变),那么参数应该声明为 `const` 以保证它不被意外修改。

如果试图修改常量类型的值,则编译器会捕捉到这个错误并发出警告或者错误消息(取决于特定的编译器)。

**软件工程知识 5.3** 当使用传值调用时,只有一个值在主调函数中被改变。这个值通过函数返回值进行赋值。如果要在主调函数中修改多个值,就应该按引用传递多个参数。

**良好编程习惯 5.3** 在使用函数之前,首先要检查这个函数的原型以确定可修改的参数。有 4 种方法可以将指针传递给函数:非常量数据的非常量指针、常量数据的非常量指针、非常量数据的常量指针和常量数据的常量指针。每一种组合提供了不同的访问权限。

非常量数据的非常量指针提供了最高的访问权限,数据可以通过复引用指针来修改,并且指针可以修改为指向其他数据。非常量数据的非常量指针的声明中不包含 `const` 限定符。这种指针可以被用来接收函数中的字符串,用指针算法处理或修改字符串中的每一个字符。图 5.10 中的函数 `convertToUppercase` 声明参数 `sPtr(char * sPtr)` 为非常量数据的非常量指针,函数用指针算法逐个处理字符串 `s` 中的字符。函数 `islower` 读取一个字符参数,如果它是小写字母或者是假值,将返回真值。字符串中 'a' 到 'z' 的字符通过函数 `toupper` 都可以转换成相对应的大写字母,其余字符不变。函数 `toupper` 取一个字符作为参数,如果字符是小写字母,就返回到对应的大写字母;否则返回原字符。函数 `toupper` 和 `islower` 是字符处理库中的一部分(参见第 16 章)。

```
1 //Fig. 5.10: fig05_10.cpp
2 //Converting lowercase letters to uppercase letters
3 //using a non-constant pointer to non-constant data
4 #include <iostream>
5
6 using std::cout;
```

```

7  using std::endl;
8
9  #include <cctype>
10
11 void convertToUppercase( char * );
12
13 int main()
14 {
15     char string[] = "characters and $32.98";
16
17     cout << "The string before conversion is: " << string;
18     convertToUppercase( string );
19     cout << " \nThe string after conversion is: "
20          << string << endl;
21     return 0;
22 }
23
24 void convertToUppercase( char * sPtr )
25 {
26     while ( *sPtr != '\0' ) {
27
28         if ( islower( *sPtr ) )
29             *sPtr = toupper( *sPtr ); //convert to uppercase
30
31         ++sPtr; //move sPtr to the next character
32     }
33 }

```

输出结果:

```

The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98

```

图 5.10 将字符串转换为大写字母

常量数据的非常量指针,指针被修改后可以指向任何适当类型的数据项,但是被指向的数据不能通过指针修改。这样的指针可以用来接收函数的数组参数,函数可以不经修改数据就可以处理每一个数组元素。例如:图 5.11 中的函数 `printCharacters` 声明参数 `sPtr` 为 `const char *` 类型,这个声明从右到左表示“`sPtr` 是字符常量的指针”。函数体使用 `for` 结构输出字符串中的每一个字符,直到遇到空中止符。每一个字符被打印出来后,指针 `sPtr` 递增,指向字符串中的下一个字符。

```

1  //Fig. 5.11: fig05_11.cpp
2  //Printing a string one character at a time using
3  //a non - constant pointer to constant data
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void printCharacters( const char * );

```



```

10
11 int main()
12 |
13     char string[] = "print characters of a string";
14
15     cout << "The string is;\n";
16     printCharacters( string );
17     cout << endl;
18     return 0;
19 |
20
21 //In printCharacters, sPtr cannot modify the character
22 //to which it points. sPtr is a "read-only" pointer
23 void printCharacters( const char *sPtr )
24 |
25     for ( ; *sPtr != '\0'; sPtr ++ )    //no initialization
26         cout << *sPtr;
27 |

```

输出结果:

```

The string is:
print characters of a string

```

图 5.11 使用常量数据的非常量指针打印字符串(一次打印一个字符)

图 5.12 演示了一个语法错误示例:函数接收常量数据的非常量指针,并试图通过指针修改数据在编译时产生的语法错误消息。

```

1 //Fig. 5.12: fig05_12.cpp
2 //Attempting to modify data through a
3 //non-constant pointer to constant data.
4 #include <iostream>
5
6 void f( const int * );
7
8 int main()
9 |
10     int y;
11
12     f( &y );    //f attempts illegal modification
13
14     return 0;
15 |
16
17 //xPtr cannot modify the value of the variable
18 //to which it points
19 void f( const int *xPtr )
20 |
21     *xPtr = 100; //cannot modify a const object
22 |

```

Borland C++ 命令行编译器输出的错误消息:

```
Error E2024 Fig05_12.cpp 20: Cannot modify a const object in
function f(const int *)
Warning W8057 Fig05_12.cpp 21: Parameter 'xPtr' is never used in
function f(const int *)
```

Microsoft Visual C++ 编译器输出的错误消息:

```
Fig05_12.cpp(20): error C2166: l-value specifies const object
```

图 5.12 试图通过常量数据的非常量指针修改数据

正如我们所知,数组是累计数据类型,用同一名称存放相同类型的相关数据项。第6章将介绍另一种名为“结构”(有些语言称之为记录)的累计数据类型。结构可以在同一个名称下存放不同数据类型的相关数据项(例如,存放公司每一名员工的信息)。调用带数组参数的函数时,这个数组自动地通过模拟按引用调用传递给函数。然而,结构总是通过传值调用传递整个结构的副本。这需要复制结构中每个数据项目,并将其存放在计算机函数堆栈中的执行时开销(当函数在执行过程中,使用函数堆栈存放函数调用中使用的局部变量)。当结构数据必须传递给函数时,可以用常量数据的指针(或常量数据的引用)得到按引用调用的性能和传值调用对数据的保护。传递结构的指针时,只要复制存放结构的地址。在4字节地址的机器上,只要复制4字节内存,而不是复制结构的几百或几千字节。

**性能提示 5.1** 使用常量数据的指针,或引用常量数据传递结构之类的大对象,能够得到按引用调用的性能优势和传值调用的安全性。

非常量数据的常量指针总是指向相同的内存地址,并且在这个地址中,数据可以通过指针修改。这里的数组名是默认的。数组名是数组开头的常量指针,所有数组中的数据都可以用数组名和数组下标访问和修改。非常量数据的常量指针可以接收数组为函数参数,该函数只用数组下标符号访问数组元素。声明为 `const` 的指针必须初始化(如果是函数参数,它应该通过传递给函数的指针初始化)。图 5.13 中的程序试图修改常量指针。指针 `ptr` 的类型声明为 `int * const`。图中的声明从右向左表示“`ptr` 是整数的常量指针”,这个指针用整数变量 `x` 的地址初始化。程序试图将地址 `y` 赋为 `ptr`,因而产生了一条错误消息。注意:当数值 7 赋给 `*ptr` 时,不产生错误,说明 `ptr` 指向的值是可修改的。

```
1 //Fig. 5.13: fig05_13.cpp
2 //Attempting to modify a constant pointer to
3 //non-constant data
4 #include <iostream>
5
6 int main()
7 {
8     int x, y;
9
10    int * const ptr = &x; //ptr is a constant pointer to an
11                          //integer. An integer can be modified
12                          //through ptr, but ptr always points
13                          //to the same memory location.
14    *ptr = 7;
15    ptr = &y;
```

```

16
17     return 0;
18 |

```

Borland C++ 命令行编译器输出的错误消息:

```
Error E2024 Fig05_13.cpp 15: Cannot modify a const object in function main()
```

Microsoft Visual C++ 编译器输出的错误消息:

```
Fig05_13.cpp(15): error C2166: l-value specifies const object
```

图 5.13 打算把常量指针修改为非常量数据

**常见编程错误 5.6** 声明为 const 的指针不在声明时初始化是语法错误。

常量数据的常量指针的访问权限最低。这样的指针总是指向相同的内存地址,在这个内存地址里的数据不能用指针修改。这就是数组如何被传递到函数,而这个函数仅用数组下标符号读取,而不能修改这个数组。图 5.14 中的程序声明指针变量 ptr 为 const int \* const 类型。这个声明从右向左表示“ptr 是常量整数的常数指针”。此图演示了当试图修改 ptr 指向的数据时和试图修改存放在指针变量里的地址时产生的错误消息。注意:当试图输出指针 ptr 指向的值时不会有错误产生,因为在输出语句中没有进行修改。

```

1 //Fig. 5.14: fig05_14.cpp
2 //Attempting to modify a constant pointer to
3 //constant data.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 int main()
10 {
11     int x = 5, y;
12
13     const int *const ptr = &x; //ptr is a constant pointer to a
14                                //constant integer. ptr always
15                                //points to the same location
16                                //and the integer at that
17                                //location cannot be modified.
18     cout << *ptr << endl;
19     *ptr = 7;
20     ptr = &y;
21
22     return 0;
23 |

```

Borland C++ 命令行编译器输出的错误消息:

```
Error E2024 Fig05_14.cpp 19: Cannot modify a const object in
function main()
```

```
Error E2024 Fig05_14.cpp 20: Cannot modify a const object in
function main()
```

Microsoft Visual C++ 编译器输出的错误消息:

```
Fig05_14.cpp(19): error C2166: 1-value specifies const object
Fig05_14.cpp(20): error C2166: 1-value specifies const object
```

图 5.14 打算把常量指针改为常量数据

## 5.6 使用引用调用的冒泡排序

下面我们使用(参见图 5.15)两个函数——bubbleSort 和 swap 修改图 4.16 中的冒泡排序程序。函数 bubbleSort 执行数组排序,它调用函数 swap 来改变数组元素 array[j]和[j+1]。请记住,C++强制函数间的信息隐藏,所以 swap 不能访问 bubbleSort 中的各个数组元素。因为 bubbleSort 需要 swap 访问将被调换的数组元素,所以 bubbleSort 使用引用调用将每个数组元素传递给 swap,每一个数组元素的地址被准确地传递。尽管整个数组自动按引用调用传递,但各个数组元素是标量并且按照常规地按传值调用来传递。因此,bubbleSort 对 swap 调用中的每个数组元素使用地址操作符(&),例如语句

```
swap ( &array[j], &array[j+1] );
```

实施按引用调用。函数 swap 用指针变量 element1Ptr 接收 &array[j]。因为隐藏了信息,swap 不知道 array[j]的名称,但是 swap 能够把 \*element1Ptr 作为 array[j]同义词。因此,当 swap 引用 \*element1Ptr 时,实际上引用的是 bubbleSort 中的 array[j]。同样地,当 swap 引用 \*element2Ptr 时,实际上引用的是 bubbleSort 中的 array[j+1]。

```
1 //Fig. 5.15; fig05_15.cpp
2 //This program puts values into an array, sorts the values into
3 //ascending order, and prints the resulting array.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 void bubbleSort( int *, const int );
14
15 int main()
16 |
17     const int arraySize = 10;
18     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19     int i;
20
21     cout << "Data items in original order\n";
22
23     for ( i = 0; i < arraySize; i ++ )
24         cout << setw( 4 ) << a[ i ];
25
```

```

26  bubbleSort( a, arraySize );           //sort the array
27  cout << " \nData items in ascending order\n";
28
29  for ( i = 0; i < arraySize; i ++ )
30      cout << setw( 4 ) << a[ i ];
31
32  cout << endl;
33  return 0;
34 |
35
36 void bubbleSort( int *array, const int size )
37 |
38     void swap( int * const, int * const );
39
40     for ( int pass = 0; pass < size - 1; pass ++ )
41
42         for ( int j = 0; j < size - 1; j ++ )
43
44             if ( array[ j ] > array[ j + 1 ] )
45                 swap( &array[ j ], &array[ j + 1 ] );
46 |
47
48 void swap( int * const element1Ptr, int * const element2Ptr )
49 {
50     int hold = *element1Ptr;
51     *element1Ptr = *element2Ptr;
52     *element2Ptr = hold;
53 }

```

输出结果:

```

Data items in original order
  2   6   4   8  10  12  89  68  45  37
Data items in ascending order
  2   4   6   8  10  12  37  45  68  89

```

图 5.15 使用引用调用的冒泡排序

尽管 swap 函数

```

    hold = array[j];
    array[j] = array[j+1];
    array[j+1] = hold;

```

不能用,但是图 5.15 中的 swap 函数

```

    hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;

```

可获得几乎同样的结果。

大家需要注意函数 bubbleSort 的几个特性。函数首部声明 array 为 int \* array[ ] 而不是 int array[ ], 表明 bubbleSort 接收单下标数组作为参数(再次重申,这些符号可以互换)。参

数 `size` 被声明为 `const` 以保证最低权限原则。尽管参数 `size` 接收 `main` 中数值的副本并且正在修改该副本但不修改 `main` 中的值,但是 `bubbleSort` 不需要改变 `size` 就可以完成它的任务。因此 `size` 被声明为 `const` 以保证它不被修改。如果数组的长度在存放过程中被修改,那么排序算法将无法正确执行。

`bubblesort` 函数体包含函数 `swap` 的原型,因为它是调用 `swap` 的惟一函数。将原型置于 `bubblesort` 中,使得只能从 `bubblesort` 正确调用 `swap`。其他试图调用 `swap` 的函数无法访问正确的函数原型,这样通常会导致语法错误,因为 C++ 需要函数原型。

**软件工程知识 5.4** 将函数原型放入其他函数的定义中能保证最低权限原则,只能从该原型所在函数中正确调用。

注意,函数 `bubbleSort` 接收数组长度参数,函数必须知道数组长度才能排序数组,这个数组长度必须被单独传递到函数。当数组传递到函数时,函数接收第一个数组元素的内存地址。

通过定义函数 `bubbleSort` 以接收数组长度为参数,我们在排序任何长度单下标整型数组的程序中使用函数。

**软件工程知识 5.5** 当把数组传递给函数时,同时传递了数组的长度(而不是在函数中建立数组长度信息),这使函数更为一般化,以便在很多程序中反复使用函数。

如果数组长度已经直接编入函数中,这就会把函数的使用限制在特定长度的数组,并且减少它的复用性。只有运行处理特定长度的单下标整型数组的程序才可以调用这个函数。

在编译过程中,C++ 提供了一元操作符 `sizeof`,用于确定数组(或者其他数据类型)的长度。在图 5.16 所示的程序中,当 `sizeof` 出现在一个数组前面,`sizeof` 操作符返回数组中的总字节数为 `size_t` 类型的值,通常是 `unsigned int` 类型。这里我们的计算机会在 4 字节内存中存放 `float` 类型变量,同时 `array` 声明为第 20 个元素,所以 `array` 使用了内存中 80 字节空间。在接收数组参数的函数中采用指针参数时,`sizeof` 操作符返回指针长度的字节数(4),而不是数组长度。

```

1 //Fig. 5.16: fig05_16.cpp
2 //Sizeof operator when used on an array name
3 //returns the number of bytes in the array.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 size_t getSize( double * );
10
11 int main()
12 {
13     double array[ 20 ];
14
15     cout << "The number of bytes in the array is "
```

```

16         << sizeof( array )
17         << " \nThe number of bytes returned by getSize is "
18         << getSize( array ) << endl;
19
20     return 0;
21 }
22
23 size_t getSize( double *ptr )
24 {
25     return sizeof( ptr );
26 }

```

输出结果:

The number of bytes in the array is 80

The number of bytes returned by getSize is 4

图 5.16 采用数组名时, sizeof 操作符返回该数组的总字节数

**常见编程错误 5.7** 在函数中使用 sizeof 操作符来寻找数组参数长度的字节数时返回指针长度的字节数,而不是数组长度的字节数。

利用两个 sizeof 操作符的结果可确定数组中元素个数。例如数组声明

```
double realArray[22];
```

如果 double 数据类型的变量被存放在 8 个字节的内存中,数组 realArray 就包含全部 176 个字节。表达式

```
sizeof realarray /sizeof (double)
```

可以确定数组元素的个数,它确定了数组 realarray 的字节数,并且将这个值除以内存中存放一个 double 值的字节数。

图 5.17 中的程序使用 sizeof 操作符来计算个人电脑中用于存放每个标准数据类型时使用的字节数。

```

1 //Fig. 5.17: fig05_17.cpp
2 //Demonstrating the sizeof operator
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10
11 int main()
12 {
13     char c;
14     short s;
15     int i;
16     long l;
17     float f;
18     double d;

```

```

19     long double ld;
20     int array[ 20 ], *ptr = array;
21
22     cout << "sizeof c = " << sizeof c
23         << "\tsizeof(char) = " << sizeof( char )
24         << "\nsizeof s = " << sizeof s
25         << "\tsizeof(short) = " << sizeof( short )
26         << "\nsizeof i = " << sizeof i
27         << "\tsizeof(int) = " << sizeof( int )
28         << "\nsizeof l = " << sizeof l
29         << "\tsizeof(long) = " << sizeof( long )
30         << "\nsizeof f = " << sizeof f
31         << "\tsizeof(float) = " << sizeof( float )
32         << "\nsizeof d = " << sizeof d
33         << "\tsizeof(double) = " << sizeof( double )
34         << "\nsizeof ld = " << sizeof ld
35         << "\tsizeof(long double) = " << sizeof( long double )
36         << "\nsizeof array = " << sizeof array
37         << "\nsizeof ptr = " << sizeof ptr
38         << endl;
39     return 0;
40 |

```

输出结果:

```

sizeof c = 1      sizeof(char) = 1
sizeof s = 2      sizeof(short) = 2
sizeof i = 4      sizeof(int) = 4
sizeof l = 4      sizeof(long) = 4
sizeof f = 4      sizeof(float) = 4
sizeof d = 8      sizeof(double) = 8
sizeof ld = 8     sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4

```

图 5.17 sizeof 操作符确定存放标准数据类型时使用的字节数

**可移植性提示 5.3** 存放特定数据类型时使用的字节数随系统的不同而不同。在编译的程序依赖于数据类型长度并且要在几个计算机系统上运行时,用 sizeof 来确定存放这种数据类型时使用的字节数。

操作符 sizeof 可以用于任何变量名、类型名和常量值。当它用在变量名(不是数组名)或者常量值时,返回存放特定变量或常量类型所用的字节数。请注意:当提供类型名称操作数时,sizeof 使用时必须加括号。记住,sizeof 是一个操作符,不是一个函数。

**常见编程错误 5.8** 提供类型名称操作数时,如在 sizeof 操作中省略括号会出现语法错误。

**性能提示 5.2** sizeof 是编译时的一元操作符,不是执行时函数。因此,使用 sizeof 不会对执行性能造成不良影响。



## 5.7 指针表达式和指针算法

在算术表达式、赋值表达式和比较表达式中,指针是有效操作数。但是,通常并非所有用于这些表达式中的操作符都在指针变量中有效。本节将介绍指针操作数的操作符及其用法。

只有少量的算术运算可用指针。指针可以自增(++)或者自减(--),整数可以添加到指针中(+或+=),也可以从指针中减去整数(-或-=),或者一个指针减去另一个指针。

假设数组 `int v[5]` 已经声明,并且其第一个元素存放在内存地址 3000。假设指针 `vptr` 已经初始化为指向数组 `v[0]`(即 `vptr` 的值为 3000)。图 5.18 演示了 4 字节整数机器中的这种情况。注意,`vptr` 可以初始化为数组 `v` 的指针,如下所示

```
vptr = v;
vptr = &v[0];
```

**可移植性提示 5.4** 如今大多数的计算机使用 2 字节或 4 字节整数。一些较新的机器使用 8 字节整数。因为指针算法的结果与指针指向的对象长度有关,所以指针算法与机器有关。

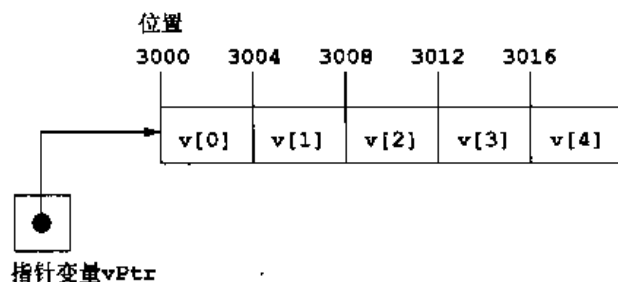


图 5.18 数组 `v` 和指向 `v` 的指针变量 `vPtr`

使用常规算法,  $3000 + 2$  的值为 3002, 这种情况通常是没有使用指针算法得出的结果。当指针增加或减少一个整数时,这个指针并不是简单地增加或减去这个整数,而是加上指针指向的对象长度的这个倍数。字节数与对象数据类型有关。以语句

```
vPtr += 2;
```

为例,假设用 4 字节内存空间存储整数时,将得出的结果为 3008 ( $3000 + 2 * 4$ )。在数组 `v` 中,`vPtr` 指向 `v[2]`(参见图 5.19)。如果整数被存储在 2 字节内存空间,那么上述计算结果将会是 3004 ( $3000 + 2 * 2$ )。如果数组是不同的数据类型,上述语句将指针递增指针所指对象长度的 2 倍。当对字符数组执行指针算法时,结果与常规算法是一样的,因为每一个字符的长度就是一个字节。

如果 `vPtr` 已经递增到 3016,指向 `v[4]`,执行语句

```
vPtr -= 4;
```

`vPtr` 将返回到 3000,即数组开头。如果指针增加 1 或减少 1,可以使用自增(++)或自减(--)操作符。语句

```
++vPtr;
vPtr ++;
```

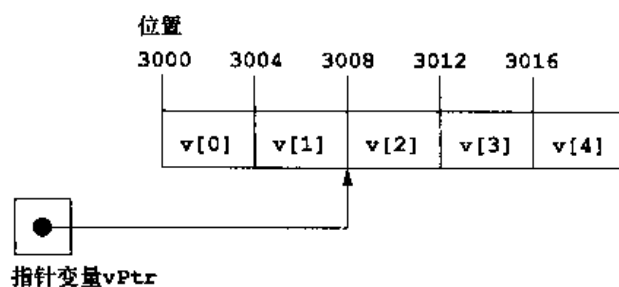


图 5.19 经过指针运算后的指针 vPtr

将指针移到数组中下一个位置。语句

```
--vPtr;
vPtr--;
```

将指针移到数组的前一个元素。

相同数组的指针变量可以相减。例如,如果 `vPtr` 包含地址 3000, `v2Ptr` 包含地址 3008, 则语句

```
x = v2Ptr - vPtr;
```

将 `x` 指定为 `vPtr` 到 `v2Ptr` 的元素个数。如果不对数组进行操作指针算法,就没有存在的必要。我们不能假设两个相同类型的指针变量存放于内存中相邻地址,除非它们是数组的相邻元素。

**常见编程错误 5.9** 对不引用数组值的指针使用指针算法是逻辑错误。

**常见编程错误 5.10** 将两个不引用相同数组元素的指针相减或相比是逻辑错误。

**常见编程错误 5.11** 使用指针算法时,超过数组边界是逻辑错误。

如果两个指针类型相同,那么一个指针能够赋给另一个指针,否则要用强制类型转换操作符将赋值语句右边的指针值转换为赋值语句左边的指针类型。这条规则有一个例外,就是 `void` 指针(也就是 `void *`),该指针是一般性指针,可以表示任何指针类型,所有指针类型都能够不经过转换赋给 `void` 指针。但是, `void` 指针不能直接赋给另一个类型的指针,而要先将 `void` 指针转换成适当的指针类型。

`void *` 指针不能复引用。例如,编译器知道 `int` 指针指向使用 4 字节整数机器中的 4 字节内存,但是 `void` 指针只是简单地包含未知数据类型的内存地址,编译器不知道指针所指的确切字节数。编译器必须要知道特定指针的数据类型才能确定该指针复引用的字节数。`void` 指针字节数不能根据类型来确定字节数。

**常见编程错误 5.12** 将一个类型的指针赋给另一个类型的指针(`void *` 类型除外),而不是先将第一个类型的指针转换成另一种类型的指针是语法错误。

**常见编程错误 5.13** 复引用 `void *` 指针是语法错误。

使用相同和相关的操作符可以比较指针,但是如果指针不指向相同数组中的元素,这种比较毫无意义。指针比较是比较指针存放的地址。例如,指向相同数组的两个指针的比较结果表明,一个指针比另一个指针所指的数组元素编号更高。指针比较常用于确定指针是否为 0。

## 5.8 指针和数组的关系

数组和指针在C++中是密切相关的,并且几乎可以互换使用。数组名可以视为常量指针,指针可以进行任何有关数组下标的操作。

**良好编程习惯 5.4** 当操作数组时,可以使用数组符号代替指针符号。尽管编译程序的时间可能会略微长一些,但程序却更加清晰。

假设声明整数数组 `b[5]` 和整数指数变量 `bPtr`。因为数组名(没有下标)是数组中第一个元素的指针,我们可以通过语句

```
bPtr = b;
```

把 `bPtr` 设置为数组 `b` 中第一个元素的地址。语句

```
bPtr = &b[0];
```

同样可以设置为数组中第一个元素的地址,数组元素 `b[3]` 还能够通过指针表达式引用

```
*(bPtr + 3)
```

上述表达式中的 3 是指针的偏移量。当指针指向数组的开头时,偏移量表示数组中的哪一个元素将被引用,并且偏移量等于数组下标。上述符号被称为指针/偏移量符号。在这里括号是必需的,因为 `*` 的优先级要高于 `+` 的优先级。如果没有括号,上述表达式将在表达式 `*bPtr` 的值中加 3(即假设 `bPtr` 指向数组的开头,3 将会被加到 `b[0]` 中)。正像数组元素能够使用指针表达式引用一样,地址

```
&b[3]
```

能够用指针表达式写为

```
bPtr + 3
```

数组名可以被视为指针并且可以在指针算法中使用。例如表达式

```
*(b + 3)
```

也引用了数组元素 `b[3]`。通常,所有加下标的数组表达式都可以写成指针加偏移量。本例中,使用指针/偏移量符号,用数组名作为指针。请注意,上述语句没有以任何方式改变数组名;`b` 仍然指向数组中第一个元素。

指针完全可以像数组一样加下标。例如表达式

```
bPtr[1]
```

指数组元素 `b[1]`;这个表达式称为指针/下标符号。

请记住,数组名本质上是个常量指针;它始终指向数组开头。因而,表达式

```
b += 3
```

是无效的,因为该表达式试图用指针算法修改数组名的值。

**常见编程错误 5.14** 尽管数组名是指向数组开头的指针,并且指针能够在算术表达式中修改,但是数组名不可以在算术表达式中修改,因为数组名实际上是个常量指针。

图 5.20 中的程序使用了前面介绍过的引用数组元素的 4 种方法,分别是:数组下标、把数组名作为指针的指针/偏移量符号、指针下标和指针的指针/偏移量符号,打印数组的 4 个元素。

```
1 //Fig. 5.20: fig05_20.cpp
2 //Using subscripting and pointer notations with arrays
3
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 int main()
10 {
11     int b[] = { 10, 20, 30, 40 }, i, offset;
12     int *bPtr = b; //set bPtr to point to array b
13
14     cout << "Array b printed with;\n"
15           << "Array subscript notation\n";
16
17     for ( i = 0; i < 4; i ++ )
18         cout << "b[" << i << "] = " << b[ i ] << '\n';
19
20
21     cout << " \nPointer/offset notation where\n"
22           << "the pointer is the array name\n";
23
24     for ( offset = 0; offset < 4; offset ++ )
25         cout << "*(b + " << offset << ") = "
26               << *( b + offset ) << '\n';
27
28
29     cout << " \nPointer subscript notation\n";
30
31     for ( i = 0; i < 4; i ++ )
32         cout << "bPtr[" << i << "] = " << bPtr[ i ] << '\n';
33
34     cout << " \nPointer/offset notation\n";
35
36     for ( offset = 0; offset < 4; offset ++ )
37         cout << "*(bPtr + " << offset << ") = "
38               << *( bPtr + offset ) << '\n';
39
40     return 0;
41 }
```

输出结果:

```
Array b printed with;
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
```

```

*(b+0)=10
*(b+2)=20
*(b+2)=30
*(b+3)=40

Pointer subscript notation
bPtr[0]=10
bPtr[1]=20
bPtr[2]=30
bPtr[3]=40

Pointer/offset notation
*(bPtr+0)=10
*(bPtr+1)=20
*(bPtr+2)=30
*(bPtr+3)=40

```

图 5.20 用前面介绍的 4 种方法引用数组元素

为进一步说明数组和指针的互换性,下面来看一下图 5.21 所示程序中的两个字符串复制函数 copy1 和 copy2。这两个函数都是将一个字符串复制到一个字符数组中。比较 copy1 和 copy2 两个函数原型之后可以发现,两个函数基本相同(由于数组和指针具有互换性)。这些函数的用途相同,但执行的过程却有所不同。

```

1 //Fig. 5.21; fig05_21.cpp
2 //Copying a string using array notation
3 //and pointer notation.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 void copy1( char *, const char * );
10 void copy2( char *, const char * );
11
12 int main()
13 {
14     char string1[ 10 ], *string2 = "Hello",
15         string3[ 10 ], string4[] = "Good Bye";
16
17     copy1( string1, string2 );
18     cout << "string1 = " << string1 << endl;
19
20     copy2( string3, string4 );
21     cout << "string3 = " << string3 << endl;
22
23     return 0;
24 }
25
26 //copy s2 to s1 using array notation
27 void copy1( char *s1, const char *s2 )
28 {

```

```

29     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i ++ )
30         ;    //do nothing in body
31     }
32
33     //copy s2 to s1 using pointer notation
34 void copy2( char *s1, const char *s2 )
35 {
36     for ( ; ( *s1 = *s2 ) != '\0'; s1 ++, s2 ++ )
37         ;    //do nothing in body
38 }

```

输出结果:

```

string1 = Hello
string3 = Good Bye

```

图 5.21 数组和指针符号复制字符串

函数 `copy1` 用数组下标符号将 `s2` 中的字符串复制到字符数组 `s1`。函数声明一个作为整数下标的整型计数器变量 `i`。For 结构的首部进行全部复制操作,而它的结构体本身是空结构。首部指定 `i` 初始化为 0,并在每次循环时加 1。for 语句的条件“`(s1[i] = s2[i]) != '\0'`”,执行从字符 `s2` 到 `s1` 的复制操作。当在 `s2` 中遇到空中止符时,它被赋给 `s1` 同时循环即告中止,因为空中止符和 `'\0'` 是相同的。请记住赋值语句中的值就是被赋给左边参数的值。

函数 `copy2` 用指针和指针算法将 `s2` 中的字符串复制到字符数组 `s1`。同样,在 for 结构首部执行整个复制操作。首部没有任何变量初始化。和 `copy1` 中一样,条件 `( *s1 = *s2 ) != '\0'` 执行复制操作。复引用指针 `s2`,产生的字符赋给复引用指针 `s1`。在条件中赋值后,指针分别移到指向数组 `s1` 中的下一个元素和字符串 `s2` 中的下一个字符。当在 `s2` 中遇到 null 中止符时,它被赋给复引用指针 `s1`,并且循环即告中止。

注意,`copy1` 和 `copy2` 的第一个参数必须是一个足够大的数组,可以容纳第二个参数中的字符串。如若不然试图写进超越数组边界的内存地址时将会出现错误。另外,注意每个函数中的第二个参数声明为 `const char *`(常量字符串)。在两个函数中,第二个参数也复制到第一个参数,一次复制一个字符,字符始终未被修改。因此,第二个参数声明为常量值的指针,执行最低权限原则。两个函数都不需要修改第二个参数,所以也没有给这两个函数提供修改第二个参数的功能。

## 5.9 指针数组

数组可以包含指针。这样的数据结构的通常用法是构成字符串组数组,通常被称为字符串数组。字符串数组中的每项都是字符串,但是在 C++ 中,字符串实际上是第一个字符的指针。所以数组中的每项实际上就是字符串中第一个字符的指针。例如下列字符串数组 `suit` 的声明

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

可以表示一副扑克牌,声明中 `suit[4]` 表示 4 个元素的数组。声明中的 `char *` 表示数组 `suit` 中的每一个元素是“char 类型指针”。数组中的 4 个值分别为“`Hearts`”,“`Diamonds`”,“`Clubs`”和“`Spades`”。每一个值作为一个空中止符结尾的字符串被存放在内存中,空中止符

结尾的字符串也就是两个引号间的字符数不止一个字符长度。这4个字符串长度分别是7, 9, 6 和 7。尽管看起来好像是字符串被放置在数组 `suit` 中, 实际上只有指针被存放在数组中 (参见图 5.22)。每一个指针指向对应字符串中的第一个字符。因此, 尽管数组 `suit` 是固定的长度, 但是它可以访问任意长度的字符串。这种灵活性是 C++ 强大的数据结构功能的其中之一。`suit` 字符串可以被放置在一个双下标数组中。双下标数组就是每一行代表一套字符串, 每一列代表一套数组名中的一个字符。这样的数据结构必须在每一行中有固定的列的数量, 并且这个列的数量要像最大字符串数量一样大。因此当存储大量的字符串时 (大多数的字符串比最长的字符串要短一些), 浪费了相当多的内存。下一节将利用字符串数组协助表演一副扑克牌。

## 5.10 案例分析: 洗牌和发牌模拟程序

在这一节里, 我们用一个随机数发生器来开发一个洗牌与发牌模拟程序。接下来用这个程序来实现某种玩牌游戏程序。为了解决一些细小的性能问题, 我们刻意使用了未达到最佳标准的洗牌和发牌的算法。在这个练习中, 开发了更加有效的算法。

利用自上而下求精法开发一个程序, 程序可以洗 52 张牌并发 52 张牌。自上而下求精法尤其适用于解决大而复杂的问题, 比前文介绍的方法更有效。

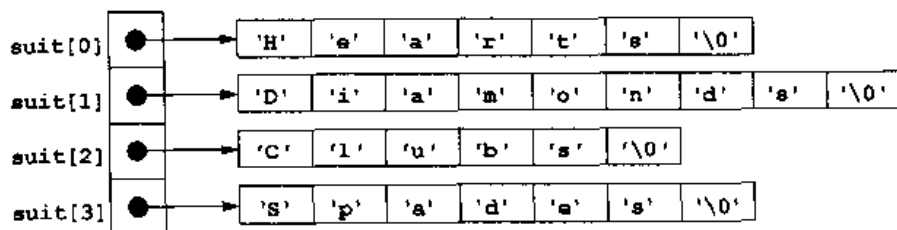


图 5.22 `suit` 数组的图形表示

我们用  $4 \times 13$  的双下标数组 `deck` 表示要玩的牌 (参见图 5.23)。行表示花色, 0 表示红桃, 1 表示方块, 2 表示梅花, 3 表示黑桃。列表示牌的面值, 0 到 9 对应 1 到 10, 10 到 12 对应 J、Q、K。我们要装入字符串数组 `suit`, 用字符串表示 4 个花色, 用字符串数组 `face` 的字符串表示 13 张牌的面值。

接下来按下面的操作模拟洗牌。首先将数组 `deck` 清空为 0, 然后随机选择 `row` (0-3) 和 `column` (0-12)。将数字插入数组元素 `deck[row][column]` 中, 它表示这个牌是洗出的牌中第一个要发的牌。这个过程将一直持续下去, 第 2 张, 第 3 张, 直到 52 被随机插入数组 `deck` 中, 表示这个牌在洗出的牌中要发的第 2、第 3、……第 52 张牌。当数组 `deck` 填上牌号时, 一张牌可能选择两次 (即当被选中的时候, `deck[row][column]` 是非 0 值)。这种选择可以被忽略, 其他 `row` 和 `column` 可以随机重复选择直到找出未被选择的牌。最后, 从 1 到 52 的值将插入数组 `deck` 中的 52 个元素中。这时, 一副扑克牌已经完全洗好。

如果已经洗过的牌重复被随机选择, 这种洗牌的算法可以执行无限长的时间。这种现象就是大家知道的无穷延迟。这里的练习中, 我们将介绍更好的洗牌算法, 它排除了无穷延迟的可能性。

|      |   |   |   |   |   |   |   |   |   |    |    |    |    |
|------|---|---|---|---|---|---|---|---|---|----|----|----|----|
|      | A | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | J  | Q  | K  |
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12 |
| 红桃 0 |   |   |   |   |   |   |   |   |   |    |    |    |    |
| 方块 1 |   |   |   |   |   |   |   |   |   |    |    |    |    |
| 梅花 2 |   |   |   |   |   |   |   |   |   |    |    |    |    |
| 梅花 3 |   |   |   |   |   |   |   |   |   |    |    |    |    |

`deck[2][12]`代表梅花K  
 梅花      King

图 5.23 双下标数组 `deck` 表示要玩的牌

有时出现在“自然”方式中的算法可能包含无穷延迟这样的微妙的性能问题。所以就要寻找避免出现无限期延迟的算法。

发第一张牌的时候,我们要寻找匹配 1 的 `deck[row][column]` 元素,这是用嵌套 for 结构进行的, `row` 取 0 到 3, `column` 取 0 到 12。那么这个数组元素对应的是哪种牌呢? `suit` 数组已经预先装入了 4 种花色,所以为了获得花色,只要打印字符串 `suit[row]`。同样,为了获得牌值,只要打印字符串 `face[column]`。我们还打印字符串“of”。按正确的顺序下打印这些信息,即可得到每张牌如“梅花 K”,“方块 A”等等。

现在用自上而下求精法进行。顶层为

```
Shuffle and deal 52 cards
```

第一步完善结果为

```
Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards
```

“Shuffle the deck”可以扩展为

```
for each of the 52 cards
  Place card number in randomly selected unoccupied slot of deck
```

“deal 52 cards”可以展开为

```
for each of the 52 cards
  find card number in deck array and print face and suit of card
```

合并得到第二步完善结果

```
Initialize the suit array
Initialize the face array
Initialize the deck array\

For each of the 52 cards
  Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards
  Find card number in deck array and print face and suit of card
```



“place card number in randomly selected unoccupied slot of deck”可以扩展为

```
Choose slot of deck randomly
While chosen slot of deck has been previously chosen
Choose slot of deck randomly
Place card number in chosen slot of deck
```

“Find card number in deck array and print face and suit of card”可以展开为

```
For each slot of the deck array
  If slot contains card number
    Print the face and suit of the card
```

合并得到第三步完善结果

```
Initialize the suit array
Initialize the face array
Initialize the deck array
For each of the 52 cards
  Choose slot of deck randomly
  While slot of deck has been previously chosen
    Choose slot of deck randomly
  Place card number in chosen slot of deck
For each of the 52 cards
  For each slot of deck array
    If slot contains desired card number
      Print the face and suit of the card
```

这就完成了完善过程。注意如果洗牌和发牌的算法组合成每张牌放到牌堆上时进行发牌,这个程序会更有效。我们选择分别编程这些操作,因为通常是先洗后发,而不是边洗边发。

图 5.24 中显示了洗牌和发牌的程序。图 5.25 显示了输出结果。注意函数 deal 所使用的输出格式

```
cout << setw(5) << setiosflags (ios::right)
      << wFace[column] << "of "
      << setw(8) << setiosflags (ios::left)
      << wSuit[row]
      << (card % 2 == 0? '\n': '\t');
```

上述输出语句使牌的面值在 5 个字符的域中以向右对齐的形式输出,而花色在 8 个字符中以向左对齐的形式输出。输出打印成两列格式。如果输出的牌在第一列,则在后面输出一个制表符,移到第二列,否则会输出换行符。

```
1 //Fig. 5.24; fig05_24.cpp
2 //Card shuffling dealing program
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8 using std::ios;
9
```

```
10 using std::setw;
11 using std::setiosflags;
12
13 #include <cstdlib>
14 #include <ctime>
15
16 void shuffle( int[][ 13 ] );
17 void deal( const int[][ 13 ], const char *[], const char *[] );
18
19 int main()
20 |
21     const char *suit[ 4 ] =
22         | "Hearts", "Diamonds", "Clubs", "Spades" };
23     const char *face[ 13 ] =
24         | "Ace", "Deuce", "Three", "Four",
25         | "Five", "Six", "Seven", "Eight",
26         | "Nine", "Ten", "Jack", "Queen", "King" |;
27     int deck[ 4 ][ 13 ] = { 0 };
28
29     srand( time( 0 ) );
30
31     shuffle( deck );
32     deal( deck, face, suit );
33
34     return 0;
35 |
36     ,
37 void shuffle( int wDeck[][ 13 ] )
38 {
39     int row, column;
40
41     for ( int card = 1; card <= 52; card ++ ) |
42         do {
43             row = rand() % 4;
44             column = rand() % 13;
45             { while( wDeck[ row ][ column ] != 0 );
46
47                 wDeck[ row ][ column ] = card;
48             }
49 |
50
51 void deal( const int wDeck[][ 13 ], const char *wFace[],
52             const char *wSuit[] )
53 {
54     for ( int card = 1; card <= 52; card ++ )
55
56         for ( int row = 0; row <= 3; row ++ )
57
58             for ( int column = 0; column <= 12; column ++ )
59
```

```

60         if ( wDeck[ row ][ column ] == card )
61             cout << setw( 5 ) << setiosflags( ios::right )
62                 << wFace[ column ] << " of "
63                 << setw( 8 ) << setiosflags( ios::left )
64                 << wSuit[ row ]
65                 << ( card % 2 == 0 ? '\n' : '\t' );
66     }

```

图 5.24 洗牌和发牌模拟程序

输出结果:

|                   |                   |
|-------------------|-------------------|
| Six of Clubs      | Seven of Diamonds |
| Ace of Spades     | Ace of Diamonds   |
| Ace of Hearts     | Queen of Diamonds |
| Queen of Clubs    | Seven of Hearts   |
| Ten of Hearts     | Deuce of Clubs    |
| Ten of Spades     | Three of Spades   |
| Ten of Diamonds   | Four of Spades    |
| Four of Diamonds  | Ten of Clubs      |
| Six of Diamonds   | Six of Spades     |
| Eight of Hearts   | Three of Diamonds |
| Nine of Hearts    | Three of Hearts   |
| Deuce of Spades   | Six of Hearts     |
| Five of Clubs     | Eight of Clubs    |
| Deuce of Diamonds | Eight of Spades   |
| Five of Spades    | King of Clubs     |
| King of Diamonds  | Jack of Spades    |
| Deuce of Hearts   | Queen of Hearts   |
| Ace of Clubs      | King of Spades    |
| Three of Clubs    | King of Hearts    |
| Nine of Clubs     | Nine of Spades    |
| Four of Hearts    | Queen of Spades   |
| Eight of Diamonds | Nine of Diamonds  |
| Jack of Diamonds  | Seven of Clubs    |
| Five of Hearts    | Five of Diamonds  |
| Four of Clubs     | Jack of Hearts    |
| Jack of Clubs     | Seven of Spades   |

图 5.25 洗牌和发牌模拟程序的输出结果

发牌算法有一个缺点。一旦找到匹配,即使已在第一次尝试中找到,内层的两个 for 结构仍然会继续搜索 deck 中的剩余元素。在随后的练习中,我们将弥补这个缺点。

## 5.11 函数指针

函数指针包含内存中函数的地址。第 4 章提到,数组名实际是数组中第一个元素的内存地址。同样,函数名实际上是执行这个函数任务的代码在内存中的开始地址。函数指针

能够传递到函数、从函数返回、存放在数组中以及赋给其他函数指针。

为了阐明函数指针的用法,我们把图 5.15 中的冒泡排序程序修改为图 5.26 中的形式。这个新程序包含 main 和函数 bubble, swap, ascending 以及 decending。函数 bubblesort 接收 ascending 或者 descending 函数指针参数以及一个整型数组和数组长度。这个程序会提示使用者来选择升序或降序来排序数组。如果使用者输入 1,则向函数 bubble 传递 ascending 函数的指针,数组将会按照升序来排列。如果使用者输入 2,则向函数 bubble 传递 descending 函数的指针,数组将会按照降序来排列。图 5.27 中显示了示例程序的输出结果。

```

1 //Fig. 5.26: fig05_26.cpp
2 //Multipurpose sorting program using function pointers
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 void bubble( int [], const int, bool ( *) ( int, int ) );
14 bool ascending( int, int );
15 bool descending( int, int );
16
17 int main()
18 |
19     const int arraySize = 10;
20     int order,
21         counter,
22         a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24     cout << "Enter 1 to sort in ascending order, \n"
25         << "Enter 2 to sort in descending order: ";
26     cin >> order;
27     cout << " \nData items in original order \n";
28
29     for ( counter = 0; counter < arraySize; counter ++ )
30         cout << setw( 4 ) << a[ counter ];
31
32     if ( order == 1 ) |
33         bubble( a, arraySize, ascending );
34         cout << " \nData items in ascending order \n";
35     |
36     else |
37         bubble( a, arraySize, descending );
38         cout << " \nData items in descending order \n";
39     |
40
41     for ( counter = 0; counter < arraySize; counter ++ )

```

```

42     cout << setw( 4 ) << a[ counter ];
43
44     cout << endl;
45     return 0;
46 }
47
48 void bubble( int work[], const int size,
49             bool ( *compare)( int, int ) )
50 {
51     void swap( int * const, int * const ); //prototype
52
53     for ( int pass = 1; pass < size; pass ++ )
54
55         for ( int count = 0; count < size - 1; count ++ )
56
57             if ( ( *compare)( work[ count ], work[ count + 1 ] ) )
58                 swap( &work[ count ], &work[ count + 1 ] );
59 }
60
61 void swap( int * const element1Ptr, int * const element2Ptr )
62 {
63     int temp;
64
65     temp = *element1Ptr;
66     *element1Ptr = *element2Ptr;
67     *element2Ptr = temp;
68 }
69
70 bool ascending( int a, int b )
71 {
72     return b < a; //swap if b is less than a
73 }
74
75 bool descending( int a, int b )
76 {
77     return b > a; //swap if b is greater than a
78 }

```

图 5.26 使用函数指针的多用途排序程序

输出结果:

```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
 2   6   4   8   10  12  89  68  45  37
Data items in ascending order
 2   4   6   8   10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

```

```
Data items in original order
  2   6   4   8  10  12  89  68  45  37
Data items in descending order
 89  68  45  37  12  10   8   6   4   2
```

图 5.27 图 5.26 中的冒泡排序程序的输出结果

bubble 函数原型中对应的参数是

```
bool (*)(int,int)
```

下面这个参数

```
bool (*compare)(int,int)
```

出现在 bubble 的函数首部中,它告诉 bubble 该参数是一个函数指针,而这个函数接收两个整型参数并且返回一个布尔运算结果。`*compare` 两边要加上括号,因为 `*` 的优先级要低于将函数参数两边所用括号的优先级。如果不添加括号,声明将如下所示

```
bool *compare(int,int)
```

它声明函数接收两个整数参数,并且返回指向布尔运算的指针。

注意,这里只包含类型,但是对于文档编制功能,程序员可以加上名称,编译器则将其忽略。

If 语句中调用传入 bubble 的函数,如下所示

```
if ( (*compare)( work[ count ], work[ count +1 ] ) )
```

就像复引用变量指针可以访问变量值一样,复引用函数指针可以执行这个函数。

也可以不经过复引用指针的过程而调用函数,如下所示

```
if ( ( compare )( work[ count ], work[ count +1 ] ) )
```

将指针直接作为函数名使用。我们更提倡通过指针调用函数的第一种方法,因为它明确地指出 `compare` 是个函数指针,通过复引用指针而调用这个函数。通过指针调用函数的第二种方法使 `compare` 看起来好像是个实际函数。这会使程序用户感到困惑,而用户希望看到函数 `compare` 的定义,结果却发现文件中没有它的定义。

函数指针的一个用法是建立菜单驱动系统。用户被提示从菜单中选择一个选项(例如:从 1 到 5)。每一个选项被不同的函数控制。指向每一个函数的指针被存放在函数指针数组中。用户选项作为数组中下标,数组中的指针是用来调用函数的。

图 5.28 中的程序提供了一个常见的声明和使用函数指针数组的例子。3 个函数——`function1`、`function2`、`function3` 已经定义,每一个函数取一个整数参数且不返回任何值。指向这 3 个函数的指针存放在数组 `f` 中,数组 `f` 按语句

```
void (*f[ 3 ])( int ) = {function1, function2, function3 };
```

进行声明,该声明从最左边的括号开始读起“是 3 个函数指针的数组,每一个函数指针取一个 `int` 参数,并且返回 `void`。”数组 3 个函数名(指针)初始化。当用户输入到 0~2 之间的值,这个值是作为函数指针数组下标。函数调用按语句

```
(f*[choice])(choice);
```

执行。调用时,`f[choice]` 选择位于数组存储地址 `choice` 的指针。复引用指针调用函数,`choice` 作为参数被传递到函数。每一个函数打印它的参数值和函数名来说明函数被正确地引用。在这个练习中,将开发一个菜单驱动系统。

```
1 //Fig.5.28: fig05_28.cpp
```

```
2 //Demonstrating an array of pointers to functions
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 void function1( int );
10 void function2( int );
11 void function3( int );
12
13 int main()
14 {
15     void ( *f[ 3 ] )( int ) = { function1, function2, function3 };
16     int choice;
17
18     cout << "Enter a number between 0 and 2, 3 to end: ";
19     cin >> choice;
20
21     while ( choice >= 0 && choice < 3 ) {
22         ( *f[ choice ] )( choice );
23         cout << "Enter a number between 0 and 2, 3 to end: ";
24         cin >> choice;
25     }
26
27     cout << "Program execution completed." << endl;
28     return 0;
29 }
30
31 void function1( int a )
32 {
33     cout << "You entered " << a
34         << " so function1 was called\n\n";
35 }
36
37 void function2( int b )
38 {
39     cout << "You entered " << b
40         << " so function2 was called\n\n";
41 }
42
43 void function3( int c )
44 {
45     cout << "You entered " << c
46         << " so function3 was called\n\n";
47 }
```

输出结果:

Enter a number between 0 and 2, 3 to end: 0

You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1

```

You entered 1 so function2 was called
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called
Enter a number between 0 and 2, 3 to end: 3
Program execution completed

```

图 5.28 演示指针数组

## 5.12 字符和字符串处理概述

本节将介绍一些通用的标准库函数,它们可以使字符串处理更加简便。这里介绍的技术适合用于开发文本编辑程序、文字处理程序、桌面排版软件、计算机化排版系统和其他文本处理软件。我们在这里使用指针基础上的字符串。在本书的后面部分中会以一章的篇幅把字符串视为一个成熟对象进行描述。

### 5.12.1 字符和字符串基础

字符是C++源程序的基本组成部分。每个程序都是由一系列字符组成,当这些字符组合在一起才真正有意义,计算机将这些字符解释为一系列指令用来完成任务。程序可以包含字符常量。字符常量是表示为单引号字符的整数值。字符常量的值是机器字符集中该字符的整数值。例如, 'z' 表示 z 的整数值(在 ASCII 字符集中为 122), '\n' 表示换行符的整数值(在 ASCII 集中为 10)。字符串就是一系列被视为为单个单元的字符。字符串可以包含字母、数字和不同的特殊字符,例如: +, -, \*, /, \$。字符串直接量或字符串常量在C++中被写在双引号中,如下所示

|                         |        |
|-------------------------|--------|
| "ohn Q.Doe"             | (名称)   |
| "9999 Main Street"      | (街道)   |
| "Wsltham,Massachusetts" | (州)    |
| "(2001)555 -1212"       | (电话号码) |

C++ 中的字符串是以空字符('\0')结尾的字符数组。通过字符串中第一个字符的指针访问字符串。字符串的值是字符串中的第一个字符的(常量)地址,因此在C++中,更确切的说法是,字符串是个常量指针,实际上也就是指向字符串第一个字符的指针。从这个意义上讲,字符串就像数组一样,因为数组名也是指向数组中第一元素的(常量)指针。

声明中可以将字符串赋给字符数组或 char \* 类型的变量。声明

```

char color [] = "blue";
const char *colorPtr = "blue";

```

分别将变量初始化为字符串"blue"。第一个声明生成 5 个元素的数组 color,包含字符'b'、'l'、'u'、'e'和'\0'。第二个声明生成指针变量 colorPtr,指向内存中的字符串"blue"。

**可移植性提示 5.5** 当用字符串直接量初始化 char \* 类型的变量时,一些编译器将字符串放在内存中无法修改字符串的位置。如果需要修改字符串直接量,则应将字符串直接量存储在字符数组中以确保在所有的系统中修改。



声明 `char color[] = "blue"` 也可在写成:

```
char color[] = ('b','l','u','e' and '\0');
```

当声明字符数组包含字符串时,数组必须要足够大以保证能够存储字符串及中止字符串的空字符。上述声明自动根据初始化值列表中提供的初始化值的个数来确定数组的长度。

**常见编程错误 5.15** 字符数组中没有分配充分的空间存储中止字符串的空字符。

**常见编程错误 5.16** 生成或使用“字符串”,不包含中止字符串的空字符。

**良好编程习惯 5.5** 将字符串存储在字符数组中时,要确保数组足够大,可以存放将要存储的最大的字符串。C++ 允许存储任意长度的字符串。如果字符串的长度大于字符串中存储的字符数组长度,超过数组边界的字符将会改写数组后面内存中的数据。

可以用 `cin` 通过流式读取字符串赋值给数组。例如:下面的语句

```
cin >> word;
```

可以用来将字符串赋值给字符数组 `word[20]`,用户输入的字符串被存储在 `word` 中。上述语句读取字符,直到遇到空格、制表符、换行符或文件结束说明符。请注意,字符串的长度不能超过 19 个字符以便留出中止字符串的空字符的空间。第 2 章介绍的 `setw` 流操作符可用于确保读取到 `word` 的字符串不超过字符数组长度。例如,语句

```
cin >> setw(20) >> word;
```

指定 `cin` 最多读取 19 个字符到数组 `word` 中,并且将数组中第 20 个位置用于存储中止字符串的空字符。`setw` 流操作符只能放在输入下一个值之前。

某些情况下,也可将一整行文本输入数组。为此,C++ 提供了函数 `cin.getline`。函数 `cin.getline` 取 3 个参数,分别是存放该行文本的字符数组、长度和分隔符。例如程序段

```
char sentence[80];
cin.getline(sentence,80, '\n');
```

声明 80 个字符的数组 `sentence`,然后从键盘读取一行文本到该数组中。遇到分隔符 `'\n'`、输入文件结束说明符或者读取的字符数比第二个参数中指定长度少 1 时,函数即停止读取字符。(保留数组中最后一个字符是用来存放中止数组的空字符。)如果遇到分隔符,读取并忽略该分隔符。函数 `cin.getline` 的第 3 个参数默认值为 `'\n'`。因此上述函数调用可写为

```
cin.getline(sentence,80);
```

第 11 章将详细介绍 `cin.getline` 和其他输入/输出函数。

**常见编程错误 5.17** 将单个字符作为字符串进行处理将导致致命的运行时错误。字符串是指针,它很可能是一个相当大的整数。但是字符只是一个小整数(0~255 之间的 ASCII 值)。在许多系统中,这样做会引发错误,因为保留低内存地址作特殊用途。例如,如果操作系统中断处理程序,将发生“访问无效”的错误。

**常见编程错误 5.18** 在需要字符作参数时,将字符作为参数传递给函数,可能导致致命的运行时错误。

**常见编程错误 5.19** 在需要字符串参数时,将字符串作为参数传递给函数是语法错误。

## 5.12.2 字符串处理库中的字符串操作函数

字符串处理库为操作字符串数据、比较字符串、搜索字符串中的字符和其他字符串、将字符串标记化(将字符串分解成逻辑组件)以及确定字符串长度提供了许多有用的函数。这一部分将介绍一些字符串处理库(标准库)中常用的字符串操作函数。图 5.29 总结了这些函数。

| 函数原型及函数说明                                                           |                                                                                                                                                                     |
|---------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char *strcpy(char *s1, char *s2)</code>                       | 将字符串 s2 复制到字符数组 s1 中, 返回 s1 的值                                                                                                                                      |
| <code>char *strncpy(char *s1, const char *s2, size_t n);</code>     | 将字符串 s2 中最多 n 个字符复制到字符数组 s1 中, 返回 s1 的值                                                                                                                             |
| <code>char *strcat(char *s1, const char *s2);</code>                | 将字符串 s2 添加到字符串 s1。s2 的第一个字符被改写为中止 s1 的空字符。返回 s1 的值                                                                                                                  |
| <code>char *strncat(char *s1, const char *s2, size_t n);</code>     | 将字符串 s2 中最多 n 个字符添加到字符串 s1 中。s2 中的第一个字符改写为中止 s1 的空字符。返回 s1 的值                                                                                                       |
| <code>int strcmp(const char *s1, const char *s2);</code>            | 比较字符串 s1 和字符串 s2。函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0、大于 0 的值                                                                                                            |
| <code>int strncmp(const char *s1, const char *s2, size_t n);</code> | 比较字符串 s1 和字符串 s2 直到第 n 个字符。函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0、大于 0 的值                                                                                                  |
| <code>char *strtok(char *s1, const char *s2);</code>                | 用一系列的 strtok 调用将字符串 s1 标记化(将字符串分成各个逻辑组件, 例如: 一行文本中的每个单词), 用字符串 s2 包含的字符将其分隔开。第一个调用包含 s1 为第一个参数, 接下来的调用继续标记化相同的字符串, 包含空中止符为第一个参数。每次调用返回当前标记的指针。如果调用函数时不再有标记, 则返回空中止符 |
| <code>size_t strlen(const char *s);</code>                          | 确定字符串 s 的长度。返回空中止符的空字符前的字符数                                                                                                                                         |

图 5.29 字符串处理库中的字符串操作函数

注意, 图 5.29 中的一些函数包含数据类型为 `size_t` 的参数。这种类型在头文件 `<stddef.h>` (`<stddef.h>` 是标准库中的头文件, 标准库还包含其他很多头文件, 例如: `<cstring.h>`) 中被定义为 `unsigned` 整数类型, 如: `unsigned int` 或者 `unsigned long`。

**常见编程错误 5.20** 使用字符串处理库中的函数时, 不包含 `<cstring.h>` 头文件。

函数 `strcpy` 将它的第二个参数(字符串)复制到它的第一个参数中。字符数组必须足够大, 可以存储字符串以及存储同样被复制的中止字符串的空字符。除了 `strncpy` 指定从字符串复制到数组中的字符数之外, 函数 `strncpy` 和 `strncpy` 是相同的。请注意, 函数 `strncpy` 不需要复制它的第二个参数的中止空字符(中止空字符只有在要被复制的字符数至少比字符串长度大 1 的时候才被写出)。例如, 如果“test”是第二个参数, 中止空字符只有在 `strncpy` 的第 3 个参数至少是 5 个字符数时才被写出(4 字符“test”加一个中止空字符)。如果第 3 个参数的字符数超过 5 个, 则空中止符就会添加到数组后面, 直到写出被第 3 个参数指定的全部字符数。

**常见编程错误 5.21** 当第 3 个参数小于或等于第二个参数的字符串长度时, 不在第一个参

数中添加中止符会导致致命的运行时错误。

图 5.30 中的程序使用 `strcpy` 将数组 `x` 中的全部字符串复制到数组 `y` 中,用 `strncpy` 将数组 `x` 中的前 14 个字符复制到数组 `z` 中。空字符 `'\0'` 被添加到数组 `z` 中,因为程序中调用 `strncpy` 并没有写入空中止符(第 3 个参数小于第 2 个参数的字符串长度)。

```

1 //Fig.5.30: fig05_30.cpp
2 //Using strcpy and strncpy
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 |
12     char x[] = "Happy Birthday to You";
13     char y[ 25 ], z[ 15 ];
14
15     cout << "The string in array x is: " << x
16         << "\nThe string in array y is: " << strcpy( y, x )
17         << '\n';
18     strncpy( z, x, 14 ); //does not copy null character
19     z[ 14 ] = '\0';
20     cout << "The string in array z is: " << z << endl;
21
22     return 0;
23 |

```

输出结果:

```

The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday

```

图 5.30 `strcpy` 和 `strncpy` 函数用法示例

函数 `strcat` 将其第二个参数(字符串)添加到它的第一个参数(字符数组包含字符串)中。第二个参数中的第一个字符代替中止第一个参数中字符串的空字符(`'\0'`)。编译器必须要保证存储第一个字符串的数组足够大,可以存储第一个字符串,第二个字符串和中止空字符(从第二个字符串复制)的总长度。函数 `strncat` 从第二个字符串添加指定的字符数到第一个字符串中。空中止符被添加到结果中。图 5.31 中的程序演示了函数 `strcat` 和函数 `strncat`。

```

1 //Fig.5.31: fig05_31.cpp
2 //Using strcat and strncat
3 #include <iostream>
4
5 using std::cout;

```

```

6  using std::endl;
7
8  #include <cstring>
9
10 int main()
11 {
12     char s1[ 20 ] = "Happy ";
13     char s2[ ] = "New Year ";
14     char s3[ 40 ] = "";
15
16     cout << "s1 = " << s1 << " \ns2 = " << s2;
17     cout << " \nstrcat(s1, s2) = " << strcat( s1, s2 );
18     cout << " \nstrncat(s3, s1, 6) = " << strncat( s3, s1, 6 );
19     cout << " \nstrcat(s3, s1) = " << strcat( s3, s1 ) << endl;
20
21     return 0;
22 }

```

输出结果:

```

s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year

```

图 5.31 strcat 和 strncat 函数用法示例

图 5.32 用 strcmp 和 strncmp 函数比较 3 个字符串。函数 strcmp 逐个比较第一个字符串参数和第二个字符串参数中的字符。如果字符串相等,函数返回 0;如果第一个字符串小于第二个字符串,函数返回负值;如果第一个字符串大于第二个字符串,函数返回正值。函数 strncmp 和 strcmp 是相等的,只是 strncmp 只比较到指定的字符数。函数 strncmp 不比较字符串中止符后面的字符。程序打印每次函数调用返回的整数值。

```

1  //Fig. 5.32: fig05_32.cpp
2  //Using strcmp and strncmp
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstring>
13
14 int main()
15 {
16     char *s1 = "Happy New Year";
17     char *s2 = "Happy New Year";

```

```

18  char *s3 = "Happy Holidays";
19
20  cout << "s1 = " << s1 << "\ns2 = " << s2
21      << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
22      << setw( 2 ) << strcmp( s1, s2 )
23      << "\nstrcmp(s1, s3) = " << setw( 2 )
24      << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
25      << setw( 2 ) << strcmp( s3, s1 );
26
27  cout << "\nstrncmp(s1, s3, 6) = " << setw( 2 )
28      << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
29      << setw( 2 ) << strncmp( s1, s3, 7 )
30      << "\nstrncmp(s3, s1, 7) = "
31      << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
32  return 0;
33  }

```

输出结果:

```

s1 =Happy New Year
s2 =Happy New Year
s3 =Happy Holidays

strcmp(s1,s2) =  0
strcmp(s1,s3) =  1
strcmp(s3,s1) = -1

strncmp(s1,s3,6) =  0
strncmp(s1,s3,7) =  1
strncmp(s3,s1,7) = -1

```

图 5.32 使用 strcmp 和 strncmp 函数

**常见编程错误 5.22** 假定函数 strcmp 和 strncmp 在其参数相等的情况下返回 1 是逻辑错误。实际运行时,在参数相等时,这两个函数返回 0(C++ 的错误值)。因此,在检测两个字符串的相等性时,函数 strcmp 和 strncmp 的结果应该与 0 相比较,以此来确定两个字符串是否相等。

为了读者便于理解一个字符串“大于”或“小于”另一个字符串的含义,我们举一姓氏字母顺序表的例子。毫无疑问,读者会把“Jones”放在“Smith”前面。因为在字母表中,“Jones”的首字母在“Smith”的首字母之前。但是字母表不仅只有 26 个字母,也是一个字母顺序表。在表中,每一个字母都有特定的位置;“Z”不仅是字母表中的一个字母,而且是字母表中的第 26 个字母。

计算机怎么知道字母的先后顺序呢?所有的字符在计算机中都表示为数字代码:当计算机比较两个字符串时,实际上是在比较字符串中字符的数字代码。

**可移植性提示 5.6** 不同的计算机表示字符的内部数字代码也有所不同。

**可移植性提示 5.7** 不要用显示测试的 ASCII 码,如“if (ch == 65)”,而要用相应的字符常量,如“if (ch == 'A’)”。

在努力实施标准化字符表示的过程中,大多数计算机生产厂家将他们的机器设计成可

以使用两种常用编码:ASCII 和 EBCDIC。ASCII 表示“美国标准交换码”,EBCDIC 表示“扩展二进制编码的十进制编码系统”。当然还有其他的编码,但是这两种是最为常用的。

ASCII 和 EBCDIC 被称为字符编码或字符集。字符串和字符操作实际上就是指操作相对应的数字编码而不是字符本身。这一点解释了C++ 中字符和小整数的互换性。由于数字代码之间存在着大于、小于和等于的关系,因此通过不同字符编码将不同字符或字符串联系起来成为可能。

函数 `strtok` 用来将字符串分解成为一系列标记,标记是一系列用分隔符(通常是空格或标点符号)分开的字符。例如,一行文本中,每一个单词可以被看作一个标记,字符间的空格可以视为分隔符。

要将字符串分解成标记,需要多次调用 `strtok` (假设字符串包含多个标记)。第一次调用 `strtok` 包含两个参数:要被标记化的字符串和其中包含用于分隔标记的字符(即分隔符)的字符串。

图 5.33 所示程序中的语句

```
tokenPtr = strtok(string, " ");
```

将 `tokenPtr` 赋值给指向 `string` 中第一个标记的指针," " 表示 `string` 中的标记被空格分开。函数 `strtok` 搜索 `string` 中不是分隔符(空格)的第一个字符,这是第一个标记的开头。然后函数搜索字符串中的下一个分隔符并将其替换空字符('\0'),这是当前标记的终点。函数 `strtok` 将标记后面的下一个指针保存到 `string` 中,并且返回当前标记的指针。

```
1 //Fig. 5.33: fig05_33.cpp
2 //Using strtok
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char string[] = "This is a sentence with 7 tokens";
13     char *tokenPtr;
14
15     cout << "The string to be tokenized is;\n" << string
16          << "\n\nThe tokens are;\n";
17
18     tokenPtr = strtok( string, " ");
19
20     while( tokenPtr != NULL ) {
21         cout << tokenPtr << '\n';
22         tokenPtr = strtok( NULL, " ");
23     }
24
25     return 0;
26 }
```

输出结果:

```
The string to be tokenized is;
This is a sentence with 7 tokens

The token are:
This
is
a
Sentence
With
7
tokens
```

图 5.33 strtok 函数用法示例

后面调用 strtok 继续将 string 标记化为 string 中包含 NULL 作为第一个参数。参数 NULL 表示调用 strtok 应该继续从上次调用 strtok 在 string 中保存的位置开始标记化。如果调用 strtok 时没有保留标记,则 strtok 返回 NULL。图 5.33 中的程序使用 strtok 将字符串“This is a sentence with 7 tokens”标记化。每一个标记被分别打印。请注意,stroke 修改输入字符串,因此,如果调用 strtok 之后再次在程序中使用字符串,则应复制这个字符串。

**常见编程错误 5.23** 没有意识到 strtok 修改了正在标记化的字符串,试图将字符串用作未改动的原始字符串。

函数 strlen 取一个字符串作为参数,并返回字符串中的字符数,字符串长度不包括中止字符串的空中止符。图 5.34 中的程序演示了函数 strlen。

```
1 //Fig. 5.34: fig05_34.cpp
2 //Using strlen
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char *string1 = "abcdefghijklmnopqrstuvwxyz";
13     char *string2 = "four";
14     char *string3 = "Boston";
15
16     cout << "The length of \"" << string1
17         << "\" is " << strlen( string1 )
18         << "\nThe length of \"" << string2
19         << "\" is " << strlen( string2 )
20         << "\nThe length of \"" << string3
21         << "\" is " << strlen( string3 ) << endl;
22
23     return 0;
24 }
```

输出结果:

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

图 5.34 strlen 函数用法示例

## 5.13 【可选案例分析】对象思想:对象间的合作

这是在学习第6章“C++面向对象编程”之前的最后一次面向对象设计任务。完成了这一部分中对象间的合作和第6章的面向对象程序技术的学习之后,就可以着手准备用C++语言编写电梯模拟程序。要完成电梯模拟程序,还需要第7章和第9章中介绍的C++技术。本节末尾提供了互联网和万维网资源列表和有关UML(统一建模语言)的参考资源。

本节重点介绍对象间的合作(交互)。当两个对象为了完成一项任务相互联系时被称为合作,对象间通过发送和接收信息进行合作。合作包括下列内容:

- (1) 谁发送类对象;
- (2) 发送什么信息;
- (3) 谁接收类对象。

第一个类对象发送的信息调用第二类的操作。第4章末尾的“对象思想”小节,确定了我们系统中的很多类操作。此处将重点介绍调用这些操作的信息。图5.35演示的是一个从4.10节提炼出来的一个类和动词短语的列表,删除了与操作不对应的动词短语。保留的动词短语是我们系统中的合作。因为已经决定链接将控制模拟程序,所以通过building类将短语“为调度程序提供时间”和“为电梯提供时间”联系在一起。由于同一个原因用building类将短语“增加时间”和“获取时间”联系在一起。

| 类              | 动词短语                                |
|----------------|-------------------------------------|
| elevator       | 将电梯按钮复位,电梯铃响,信号通知电梯到达一层,打开电梯门,关闭电梯门 |
| clock          | 计时每一秒钟                              |
| scheduler      | 告诉乘客到达一层,检证这一层没有人                   |
| person         | 按楼层按钮,按电梯按钮,进入电梯,出电梯                |
| floor          | 将电梯按钮复位,关灯,开灯                       |
| floorbutton    | 呼叫电梯                                |
| elevatorbutton | 信号通知电梯准备离开                          |
| door           | (开门)信号通知乘客离开电梯,(开门)信号通知乘客进入电梯       |
| Bell           |                                     |
| Light          |                                     |
| Building       | 增加时间,获取时间,为调用程序提供时间,为电梯提供时间         |

图 5.35 修改之后的系统中的类的动词短语表

我们验证动词表以确定它们在系统中的合作。例如,elevator类将短语“将电梯按钮复位”列表。要完成这项任务,elevator类对象必须发送resetbutton信息给elevatorbutton类对象,同时调用这个类的resetbutton操作。图5.36列出了所有能够从动词短语表中收集的合作。



| 类对象            | 发送的消息           | 接受消息的类对象       |
|----------------|-----------------|----------------|
| Elevator       | resetButton     | ElevatorButton |
|                | ringBell        | Bell           |
|                | elevatorArrived | Floor          |
|                | openDoor        | Door           |
|                | closeDoor       | Door           |
| Clock          |                 |                |
| Scheduler      | stepOntoFloor   | Person         |
|                | isOccupied      | Floor          |
| Person         | pressButton     | FloorButton    |
|                | pressButton     | ElevatorButton |
|                | passengerEnters | Elevator       |
|                | passengerExits  | Elevator       |
|                | personArrives   | Floor          |
| Floor          | resetButton     | FloorButton    |
|                | turnOff         | Light          |
|                | turnOn          | Light          |
|                | summonElevator  | Elevator       |
| FloorButton    |                 |                |
| ElevatorButton | prepareToLeave  | Elevator       |
| Door           | exitElevator    | Person         |
|                | enterElevator   | Person         |
| Bell           |                 |                |
| Light          |                 |                |
| Building       | tick            | Clock          |
|                | getTime         | Clock          |
|                | processTime     | Scheduler      |
|                | processTime     | Elevator       |

图 5.36 电梯系统中的合作

### 5.13.1 合作图

现在我们来考虑一下对象,对象间必须要交互,这样在这个模拟程序中当电梯到达一层时,乘客才可以进入或离开电梯。UML(统一建模语言)提供了合作图来模拟这种相互作用。合作图和顺序图都提供了关于对象间如何相互作用的信息,但是每一个图的侧重点都不相同。顺序图侧重于这些相互作用何时出现,而合作图侧重于哪一个对象参与相互合作。

图 5.37 中的合作图模拟当 person 类对象进入和离开电梯时,系统中对象间的相互合作。当电梯到达一层,对象间的合作开始。就像在顺序图中,合作图中的对象表示为包含对象名的矩形。

相互合作的对象用实线连接,信息沿着这些线上箭头所指的方向传递。箭头所指的是信息名。

合作图中的信息顺序按照数字从小到大的顺序排列。在这个图中,编号从信息 1 开始。电梯将信息(resetbutton)传递到电梯按钮将电梯按钮复位,然后电梯将 ringbell 信息(第 2 条信息)发送到铃声,接着电梯报告到达的一层(第 3 条信息),这样可以复位楼层的按钮并把灯打开(分别是信息 3.1 和信息 3.2)。

复位楼层按钮,打开灯之后,电梯打开门(第 4 条信息)。同时,电梯门将 exitelevator 信

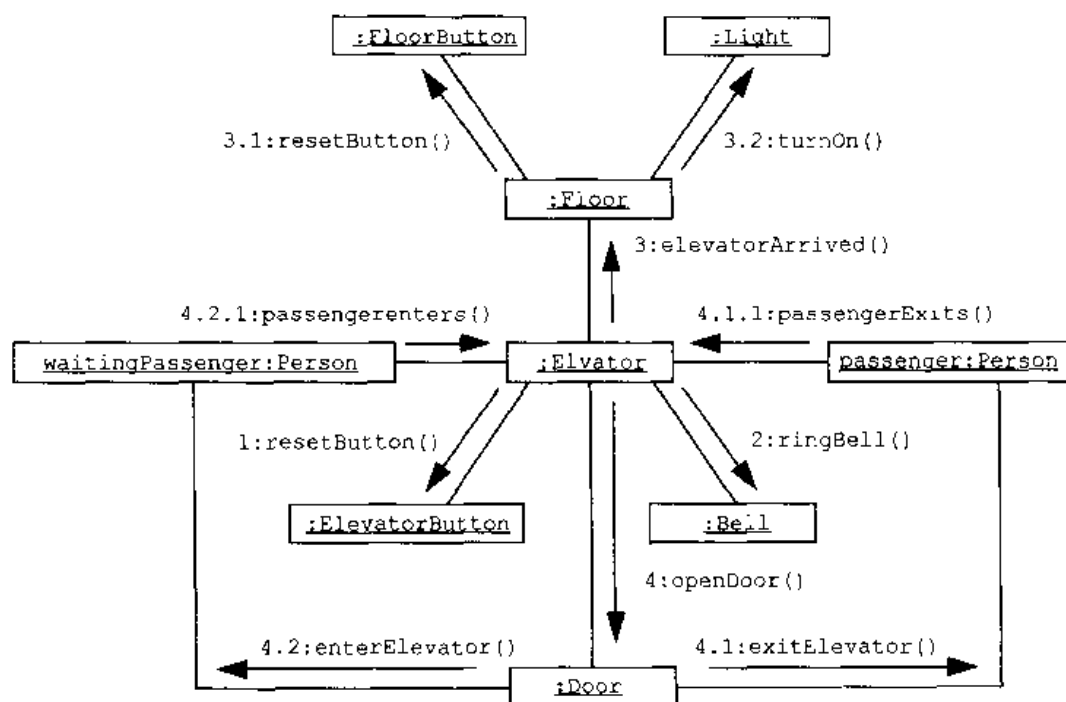


图 5.37 加载和卸载信息的合作图

息(信息 4.1)发送到对象 passenger<sup>①</sup>。对象 passenger 通过信息 passengerexits(信息 4.1.1)通知电梯将要离开。

在操作电梯离开之后,在楼层等候的乘客(waitingpassenger 对象)可以进入电梯。注意,在 passenger 对象将 passengerexits 信息发送到电梯(信息 4.1.1)之后,门将 enterelevator 信息(信息 4.2)发送到 waitingpassenger 对象。这个顺序保证了在楼层等候的乘客进入电梯之前,电梯里的乘客离开电梯。Waitingpassenger 对象通过 passengerenters 信息(信息 4.2.1)进入电梯。

### 5.13.2 小结

现在已经有有了一个相当完整的需要执行电梯模拟系统的类以及这些类之间相互作用的列表。下一章将介绍C++中的面向对象编程。学习完第6章,就可以准备写C++中电梯模拟程序的实质部分。完成第7章的学习后,就可以执行一个完整的、实用的电梯模拟程序。第9章,将描述如何使用继承开发类之间的共性,尽量减少需要执行系统的软件数。

下面把从第2~5章使用过的面向对象设计过程总结一下:

(0) 在分析阶段中,遇到客户(请你编译系统的人)并尽可能地多收集关于这个系统的信息。随着这些信息,生成一个用例,描述用户和系统相互作用的方法。在我们的案例分析中,没有将重点放在分析阶段。这一阶段的结果在问题语句中表示出来,并且用例出自这个语句。再次提醒大家注意,现实世界系统通常有许多用例。

<sup>①</sup> 现实世界中,乘电梯的人要等电梯门开之后才能离开电梯。我们必须模拟这个行为,因此设计了电梯门将信息发送到电梯中的对象 passenger 这一步骤。这条信息给电梯中的人提供了一个能看得见的指令。人接收到这个指令就会离开电梯。

(1) 通过使用问题语句中的名词列表将类放置在系统中。我们将明确表示类属性的名词和明确表示是正在被模拟的软件系统的一部分名词从表中清除。即生成一个模拟系统中类和类之间的关系(联系)的类图。

(2) 通过描述系统中每一个类的单词和短语列表,从问题语句中选出每一个类的属性。

(3) 了解更多关于系统动态属性的信息。生成一个状态图来了解类在系统中如何转换时间。

(4) 验证和每一个类相关的动词和动词短语。用这些短语读取出系统中类的操作。活动图可以帮助模拟这些操作的细节。

(5) 验证不同对象间的合作,使用顺序图和合作图模拟对象间的合作。当设计进程需要时,将属性和操作添加到类。

(6) 这时,我们的设计可能会遗漏一些组件。第6章中开始执行C++中的电梯模拟程序时,这些遗漏的组件会显得很明显。

### 5.13.3 因特网和万维网上的 UML(统一建模语言)资源

下面收集了大量因特网和万维网上关于 UML(统一建模语言)资源的参考资料。这些资源包括 UML 1.3 的说明、其他参考材料、通用资源、使用指南、常见问题解答(FAQS)、背景资料和软件。

#### 参考资料

[www.omg.org](http://www.omg.org)

该网站是 OMG(对象管理集团)的官方网站。OMG 负责 UML(统一建模语言)的维护和未来的修改工作。他们的网站包含有关 UML 和其他面向对象技术的信息。

[www.rational.com](http://www.rational.com)

瑞理软件公司(Rational Software)是最早开发 UML 的公司。该网站介绍了 UML 及其创始人:Grade Booch,James Rumbaugh 和 Ivar Jacobson。

[www.omg.org/cgi-bin/doc?ad/99-06-09](http://www.omg.org/cgi-bin/doc?ad/99-06-09)

可在此获得 PDF 和 ZIP 格式的标准 UML 1.3 说明。

[www.omg.org/techprocess/meetings/schedule/UML\\_1.4\\_RTF.html](http://www.omg.org/techprocess/meetings/schedule/UML_1.4_RTF.html)

OMG 集团在该网站维护关于 UML 1.4 说明。

[www.rational.com/uml/resources/quick/index.jhtml](http://www.rational.com/uml/resources/quick/index.jhtml)

瑞理软件公司的 UML 快速参考向导。

[www.holub.com/class/oo\\_design/uml.html](http://www.holub.com/class/oo_design/uml.html)

该网站提供了详细的带附加注释的 UML 快速参考卡片。

[softdocwiz.com/UML.htm](http://softdocwiz.com/UML.htm)

负责若干 UML 资源撰写的 Kendall Soott 负责维护该网站 UML 词典。

#### 资源

[www.omg.org/uml](http://www.omg.org/uml)

OMG 的 UML 资源页面。

[www.rational.com/uml/index.jtmpl](http://www.rational.com/uml/index.jtmpl)

瑞理软件公司的 UML 资源页面。

[www.platinum.com/corp/uml/uml.htm](http://www.platinum.com/corp/uml/uml.htm)

UML 合作开发者 Platinum Technology 在此维护 UML 资源页面。

[www.cetus-links.org/oo\\_uml.html](http://www.cetus-links.org/oo_uml.html)

该网站提供了上百个指向 UML 网站的链接,包括信息、指南和软件。

[www.uml-zone.com](http://www.uml-zone.com)

该网站提供大量的 UML 信息,包括了一些文章,还可以链接到新闻版块和其他网站。

[Home.pacbell.net/ckobryn/uml.htm](http://Home.pacbell.net/ckobryn/uml.htm)

该网站由经验丰富的 UML 软件工程师 Cris Kobryn 维护,提供常用信息,也可链接到其他重要的网站。

[www.methods-tools.com/cgi-bin/discussionuml.cgi](http://www.methods-tools.com/cgi-bin/discussionuml.cgi)

该网站提供 UML 学习板块的标题版。

[www.pols.co.uk/usecasezone/index.htm](http://www.pols.co.uk/usecasezone/index.htm)

该网站提供关于用例的资源 and 文章。

[www.ics.uci.edu/pub/arch/uml/uml\\_books\\_and\\_tools.html](http://www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html)

该网站可以链接其他 UML 书籍的信息,还包含一个支持 UML 符号的列表。

[home.earthlink.net/~salhir/](http://home.earthlink.net/~salhir/)

该网站由《UML in a Nutshell》的作者维护,可以链接到很多 UML 资源。

## 软件

[www.rational.com/products/rose/indexljtmpl](http://www.rational.com/products/rose/indexljtmpl)

这是有理软件公司的可视化 UML 建模工具 Rational Rose 的主页。可以在此下载试用版并享有免费使用期。

[www.rosearchitect.com/](http://www.rosearchitect.com/)

该网站是瑞理软件公司出版的在线杂志,内容包括使用 Rational Rose UML 建模的整个过程。

[www.advancedsw.com/](http://www.advancedsw.com/)

Advanced Software Technologies 是 GDPro(一个可视化的 UML 建模工具)的开发者,可在该网站下载试用版并享有一定的免费试用期。

[www.visualobject.com/](http://www.visualobject.com/)

Visual Object Modelers 已开发出一个可视化的 UML 建模工具。你可从该网站下载试用版,并享有一定的免费试用期。

[www.microgold.com/version2/stage/product.html](http://www.microgold.com/version2/stage/product.html)

Microgold 软件公司开发了 WithClass,这是一个支持 UML 符号的应用设计软件。

[www.lysator.liu.se/~alla/dia.html](http://www.lysator.liu.se/~alla/dia.html)

dia 是可以画出 UML 和类图表的 gtk + 图表工具。Dia 在 UML 下运行。该网站提供指向 Windows 版本的链接。

[dir.lycos.com/computers/Software/Object\\_Oriented/Nethodologies/UML/Tools/](http://dir.lycos.com/computers/Software/Object_Oriented/Nethodologies/UML/Tools/)

该网站列出了许多 UML 模拟工具及其相关主页。

[www.methods-tools.com/tools/modeling.html](http://www.methods-tools.com/tools/modeling.html)

该网站提供了一个列表,包含了很多支持 UML 的面向对象的模拟工具。

### 文章和白皮书

[www.omg.org/news/pr99/UML\\_2001\\_CACM\\_Oct99\\_p29-Kobryn.pdf](http://www.omg.org/news/pr99/UML_2001_CACM_Oct99_p29-Kobryn.pdf)

该文章由 Cris Kobryn 撰写,分析了 UML 的过去、现状以及未来。

[www.sdmagazine.com/uml/focus.rosenberg.htm](http://www.sdmagazine.com/uml/focus.rosenberg.htm)

本文提供了一些技巧,告诉大家如何将 UML 与自己的项目结合在一起。

[www.db.informatik.uni-bremen.de/umlbib/](http://www.db.informatik.uni-bremen.de/umlbib/)

UML 文献目录中提供了很多 UML 文章的标题及作者。大家可以按作者或标题搜索文章。

[usecasehelp.com/wp/white\\_papers.htm](http://usecasehelp.com/wp/white_papers.htm)

该网站提供了有关将用例模拟应用到系统分析和设计中的背景资料列表。

[www.ratio.co.uk/white.html](http://www.ratio.co.uk/white.html)

该网站有许多关于如何用 UML 进行 OOAD 进程提纲的背景资料,还包括 C++ 中的一些执行程序。

[www.tucs.fi/publications/techreports/tr\\_234.pdf](http://www.tucs.fi/publications/techreports/tr_234.pdf)

该文件夹包括一个使用 UML 的数字声音记录器的 OOAD 案例学习。

[www.sdmagazine.com/](http://www.sdmagazine.com/)

软件开发在线杂志提供了 UML 文章信息库,你可根据题目和文章标题进行搜索。

### 教程

[www.qoses.com/education/](http://www.qoses.com/education/)

该网站提供了 UML 作者 Kendall Scott 创建的一系列学习教程,由 Qoses 维护。

[www.rational.com/products/rose/tryit/tutorial/index.jtmpl](http://www.rational.com/products/rose/tryit/tutorial/index.jtmpl)

瑞理软件公司在此提供了 Rational Rose 使用指南。

### 常见问题解答

[www.rational.com/uml/gstart/faq.jtmpl](http://www.rational.com/uml/gstart/faq.jtmpl)

瑞理软件公司的 UML 常见问题解答网站。

[www.jguru.com/jguru/faq/](http://www.jguru.com/jguru/faq/)

在搜索框输入 UML,即可访问该网站的 UML 常见问题解答。

[www.uml-zone.com/umlfaq.asp](http://www.uml-zone.com/umlfaq.asp)

该网站包含一个小的 UML 常见问题解答文件夹,由 [uml-zone.com](http://uml-zone.com) 维护。

## 5.14 小结

- 指针变量的值为内存地址。

- 声明

```
int *ptr;
```

含义为声明变量 ptr 是 int 类型的指针变量,或者说成“ptr 是 int 的指针”。声明中的 \* 用于指定变量是指针变量。

- 指针可以被初始化为 0、NULL 或同类型对象的地址。将指针初始化为 0 和初始化为 NULL 是一样的。
- 0 是惟一可不经过转换赋值给指针变量的整数。
- &(地址)操作符返回操作数的地址。
- 地址操作符的操作数必须是变量名(或者一个 lvalue);地址操作符不能用于常量、不返回 lvalue 的表达式和存储类 register 声明的变量。
- \* 操作符通常被称为间接操作符或复引用操作符。返回操作数所指对象名的同义词、别名或译名。这被称为复引用指针。
- 用参数调用函数,主调函数需要被调函数修改这个参数时,传递该参数地址。然后被调函数使用间接操作符(\*)修改调用函数中的参数值。
- 接收地址参数的函数必须要有一个与其相对应的参数作为指针。
- 在函数原型中不需要包括指针名,只需要包括指针类型。指针名可以用于文档说明,而编译器则忽略指针名。
- 限定符 const 使程序员通知编译器,特定变量的值不能修改。
- 如果试图修改 const 变量值,编译器会捕捉这个错误并发出警告或错误消息(取决于特定的编译器)。
- 将指针传递给函数有 4 种方法:非常量数据的非常量指针、常量数据的非常量指针、非常量数据的常量指针和常量数据的常量指针。
- 数组自动用指针按引用传递,因为数组名的值为数组第一元素的地址。
- 要用指针按引用调用传递数组中的单个元素,就必须传递特定数组元素的地址。
- C++ 提供特殊的一元操作符 sizeof,确定程序执行期间数组的长度或其他数据类型长度(字节数)。
- 采用数组名时,sizeof 操作符数组中全部的字节数作为整数返回。
- 操作符 sizeof 可以用于任何变量名、任何类型和任何常量。
- 指针可以使用自增(++ )或自减(-- )操作符,指针中可以加进整数(+ 或 +=)、也可以减去整数(- 或 -=),指针可以减去另一个指针。
- 将指针增加或者减少一个整数时,指针增加或减少指针所指对象长度的这个倍数。
- 指针算法只能用于相邻内存地址,如数组。数组中的所有元素在内存中是相邻存放的。
- 对字符数组进行指针算法时,结果与普通算法相同,因为每一个字符占一个字节内存。
- 如果两个指针的类型相同,可以将指针赋给另一个指针。否则,必须要进行一个类型转换。除非是 void 指针,该指针是一个普通指针,可以表示任何类型的指针。其他类型指针可以赋给指针 void。但是,void 指针必须要经过正确的类型转换,才能赋给另一类型的指针。

- void 指针不能复引用。
- 指针可用相等和关系操作符比较。但这种比较只适用于比较相同数组成员的指针。
- 指针可以像数组名一样带下标。
- 数组名和数组第一个元素的指针是相同的。
- 指针/偏移量符号中的偏移量和数组下标是相同的。
- 所有带有下标的数组表达式都可以通过使用将数组名作为指针或指向数组的独立指针写成指针和偏移量。
- 数组名是一个常量指针,总是指向内存中相同的地址。
- 数组可以包含指针。
- 函数指针是函数在内存中的地址。
- 函数指针可以传递到函数、从函数中返回、存放在数组中和赋给其他指针。
- 函数指针的一个通常用法是用在所谓的菜单驱动系统中。函数指针用于选择调用特定菜单选项的函数。
- 函数 strcpy 将第二个参数(字符串)复制到第一个参数(字符数组)中。程序员要保证目标数组足够大,足以存入字符串和空中止符。
- 函数 strncpy 和 strcpy 相同,不过调用 strncpy 指定从字符串复制到数组的字符数。对于空中止符,需在复制的字符数比字符串长度至少多 1 时,才可以进行复制。
- 函数 strcat 将第二个字符串参数(包括空中止符)添加到第一个字符串参数中。第二个字符串中的第一个字符代替了第一个字符串中的空中止符('\0')。程序员必须保证存放第一个字符串的目标数组足够大,可以存放第一个字符串和第二个字符串。
- 函数 strncat 将特定的字符数从第二个字符串添加到第一个字符串中,并在结果中添加空中止符。
- 函数 strcmp 一次一个字符地比较第一个字符串参数和第二个字符串参数。如果字符串相等,则函数返回 0;如果第一个字符串小于第二个字符串,则函数返回负值;如果第一个字符串大于第二个字符串,则函数返回正值。
- 函数 strncmp 和 strcmp 相同,不过 strncmp 只比较指定的字符数。如果字符串中的字符数小于指定的字符数,strncmp 将一直比较字符直到遇到小字符串中的空中止符。
- 函数 strtok 将字符串分解为一系列标记,标记被第二个字符串参数中包含的字符分隔开。第一次调用将要标记化的字符串作为第一个参数。再调用时,第一个参数为 NULL,继续将字符串标记化。当前标记化的指针由每次调用返回。如果调用 strtok 时不再有标记,就会返回空。
- 函数 strlen 取一个字符串作为参数,并返回其字符个数,字符串长度不包含空中止符。

## 本章术语

adding a pointer and an integer 指针与整数相加  
 address operator(&) 地址操作符 &  
 appending strings to other strings  
   将字符串添加到另一个字符串中  
 array of pointers 指针数组

array of strings 字符串数组  
 ASCII 美国信息交换标准码  
 call-by-reference 引用调用  
 call-by-value 传值调用  
 character code 字符编码

character constant 字符常量  
 character pointer 字符指针  
 character set 字符集  
 class responsibilities and collaborations(CRC)  
     类、责任与合作  
 comparing strings 比较字符串  
 constant pointer 常量指针  
 constant pointer to constant data  
     常量数据的常量指针  
 constant pointer to nonconstant data  
     非常量数据的常量指针  
 copying strings 复制字符串  
 decrement a pointer 递减指针  
 delimiter 分隔符  
 dereference a pointer 复引用指针  
 dereferencing operator( \* ) 复引用操作符( \* )  
 directly reference a variable 直接引用变量  
 EBCDIC 扩充的二十进制交换码  
 function pointer 函数指针  
 increment a pointer 递增指针  
 indefinite postponement 无限延迟  
 indirection 间接  
 indirectly operator( \* ) 间接操作符 \*  
 indirectly reference a variable 间接引用变量  
 initialize a pointer 初始化指针  
 length of a string 字符串长度  
 literal 直接量  
 nonconstant pointer to constant data  
     常量数据的非常量指针  
 nonconstant pointer to nonconstant data

## “对象思想”术语

collaboration 合作  
 collaboration diagram 合作图  
 interaction among objects 对象之间的交互  
 message 信息  
 numbers in UML collaboration diagram  
     UML 合作图中的数字  
 rectangle symbol in UML collaboration diagram

## 常见编程错误

5.1 假设用于声明指针的星号( \* )可以分配给声明中用逗号分隔的,指针变量列表中的所有指针变量名,就能将指针声明为非指针。每个指针在声明时必须在名称前面加星号( \* )。

非常量数据的非常量指针  
 NULL pointer NULL 指针  
 numeric code of a character 字符的数字代码  
 offset 偏移量  
 pointer 指针  
 pointer arithmetic 指针算法  
 pointer assignment 指针赋值  
 pointer comparison 指针比较  
 pointer expression 指针表达式  
 pointer indexing 指针索引  
 pointer/offset notation 指针/偏移量符号  
 pointer subscripting 指针下标  
 pointer to a function 函数指针  
 pointer to void( void \* ) void 指针  
 pointer types 指针类型  
 principle of least privilege 最低权限原则  
 simulated call-by-reference 模拟的引用调用  
 string 字符串  
 string concatenation 字符串连接  
 string constant 字符串常量  
 string literal 字符串直接量  
 string process 字符串处理  
 subtracting an integer from a pointer  
     将指针减去一个整数  
 subtraction two pointers 两个指针相减  
 token 标记  
 tokenizing strings 标记化字符串  
 void \* ( pointer to void ) void 指针  
 word processing 字处理

UML 合作图中的矩形符号  
 sequence of messages 信息顺序  
 solid line symbol in UML collaboration diagram  
     UML 合作图中的实线符号  
 solid line with arrowhead symbol in UML collaboration diagram  
     UML 合作图中带箭头的实线符号  
 when interactions occur 当发生交互时



- 5.2 如果指针没有初始化或没有指定指向内存中的特定地址,而复引用指针可能造成致命的运行时错误,或者意外修改重要数据,虽然允许程序运行,但得到的是错误结果。
- 5.3 复引用指针是语法错误。
- 5.4 复引用 0 指针通常是致命的运行时错误。
- 5.5 需要获得指针指向的变量值时,不复引用指针是错误的。
- 5.6 声明为 const 的指针不在声明时初始化是语法错误。
- 5.7 在函数中使用 sizeof 操作符来寻找数组参数长度的字节数时返回指针长度的字节数而不是数组长度的字节数。
- 5.8 如果提供类型名称操作数时,在 sizeof 操作中省略括号是语法错误。
- 5.9 对不引用数组值的指针使用指针算法通常是逻辑错误。
- 5.10 将两个不引用相同数组元素的指针相减或相比较通常是逻辑错误。
- 5.11 当使用指针算法时,超过数组边界通常是逻辑错误。
- 5.12 将一个类型的指针赋给另一个类型的指针(除了 void \*)不把第一种类型的指针转换或另一种类型的指针是语法错误。
- 5.13 复引用 void \* 指针是语法错误。
- 5.14 尽管数组名是指向数组开头的指针,并且指针能够在算术表达式中修改,但是数组名不可以在算术表达式中修改的,因为数组名实际上是个常量指针。
- 5.15 字符数组中没有分配充分的空间存储中止字符串的空字符。
- 5.16 生成或使用“字符串”,不包含中止字符串的空字符。
- 5.17 将单个的字符作为字符串进行处理将会导致致命的运行时错误。字符串是指针,它很可能是一个相当大的整数,但是字符只是一个小整数(0~255 之间的 ASCII 值)。在许多系统中,这样做会引发错误,因为保留低内存地址是用于特殊用途。例如,如果操作系统中断处理程序,将发生“访问无效”的错误。
- 5.18 在需要字符串时,将字符作为参数传递到函数会导致致命的运行时错误。
- 5.19 在需要字符串时,将字符作为参数传递到函数是语法错误。
- 5.20 当使用字符串处理库中的函数时,没有包含 <cstring> 头文件。
- 5.21 当第 3 个参数小于或等于第 2 个参数的字符串长度时,不在第一个参数中添加中止符会导致致命的运行时错误。
- 5.22 假设 strcmp 和 strncmp 在其参数相等的情况下返回 1 是逻辑错误。实际运行时在参数相等时,这两个函数返回 0(C++ 的错误值)。因此,在检测两个字符串的相等性时,函数 strcmp 和 strncmp 的结果应该与 0 相比较,以此来确定两个字符串是否相等。
- 5.23 没有意识到 strtok 修改了正在标记化的字符串,试图将字符串用作未修改的原始字符串。

## 良好编程习惯

- 5.1 尽管并不一定要求这样做,但是变量名中包含字母 Ptr 能够更清楚地表示这个变量是指针变量并且需要适当处理。
- 5.2 如果主调函数没有明确要求被调函数在主调函数环境中修改参数变量值,使用传值调用将参数传递到函数,这是最低权限原则的另一个例子。
- 5.3 在使用函数之前,首先要检查这个函数的原型以确定它能够修改的参数。

- 5.4 当操作数组时,可以使用数组符号代替指针符号。尽管编译程序的时间可能会略微长一些,但程序却更加清晰。
- 5.5 将字符串存储在字符数组中时,要确保数组足够大,可以存放将要存储的最大的字符串。C++ 允许存储任意长度的字符串。如果字符串的长度大于字符串中存储的字符数组长度,超过数组边界的字符将会改写数组后面内存中的数据。

### 性能提示

- 5.1 使用常量数据的指针,或引用常量数据传递结构之类的大对象,能够得到按引用调用的性能优势和传值调用的安全性。
- 5.2 sizeof 是编译时的一元操作符,不是执行时函数。因此,使用 sizeof 不会对执行性能造成不良影响。
- 5.3 有时出现在“自然”方式中的算法可能包含无限期延迟这样的微妙的性能问题,所以要寻找避免出现无限期延迟的算法。

### 可移植性提示

- 5.1 指针输出形式与机器有关。有的系统用十六进制整数,有的系统则用十进制整数。
- 5.2 尽管在 ANSI C 语言和 C++ 中已经很好的定义了 const 限定符,但有些编译器仍然无法正确实现。
- 5.3 存放特定数据类型时使用的字节数随系统的不同而不同。在编译的程序依赖于数据类型长度并且要在几个计算机系统上运行时,用 sizeof 来确定存放这种数据类型时使用的字节数。
- 5.4 如今大多数的计算机使用 2 字节或 4 字节整数,一些较新的机器使用 8 字节整数。因为指针算法的结果与指针指向的对象长度有关,所以指针算法与机器有关。
- 5.5 当用字符串直接量初始化 char \* 类型的变量时,一些编译器将字符串放在内存中无法修改字符串的位置。如果需要修改字符串直接量,则应将字符串直接量存储在字符数组中以确保在所有的系统中修改。
- 5.6 不同的计算机表示字符的内部数字代码也有所不同。
- 5.7 不要用显示测试的 ASCII 码,如“if (ch == 65)”,而要用相应的字符常量,如“if (ch == 'A')”。

### 软件工程知识

- 5.1 const 限定符可以执行最低权限原则。利用最低权限原则正确设计软件,既能显著减少调试时间和负面影响,又可简化程序的修改和维护。
- 5.2 如果传递到函数体中的值不变(或者将不改变),参数就应该声明为 const 以保证它不会被意外修改。
- 5.3 使用传值调用时,主调函数中只改变一个值,这个值通过函数返回值进行赋值。如果要在主调函数中修改多个值,就应该按引用传递多个参数。
- 5.4 将函数原型放入其他函数的定义中能保证最低权限原则,只能从该原型所在函数中正确调用。
- 5.5 当把数组传递给函数时,同时传递了数组的长度(而不是在函数中建立数组长度信息),

这使函数更为一般化,以便函数可以在很多程序中反复使用。

## 自测题

### 5.1 填空题:

- a) 指针变量包含另一个变量的\_\_\_\_\_值。
- b) 可以初始化指针的值有以下 3 个:\_\_\_\_\_、\_\_\_\_\_、或\_\_\_\_\_。
- c) 可以赋给指针的惟一整数是\_\_\_\_\_。

### 5.2 判断正误,如果不正确,请说明原因。

- a) 地址操作符 & 只能用于常量、表达式和用 register 存储类声明的变量。
- b) 声明为 void 的指针可以复引用。
- c) 不同类型的指针不必经过类型转换操作就可以相互赋值。

### 5.3 回答下列问题。假设单精度浮点数被存放在 4 字节中,数组在内存中的开始地址为 1002500。下面每道题应使用前面题的结果。

- a) 声明 double 类型数组,叫做 number,含有 10 个元素,并将 10 个元素初始化为 0.0, 1.1, 2.2, ..., 9.9。假设符号化常量 SIZE 已经定义为 10。
- b) 声明指向 double 类型对象的指针 Ptr。
- c) 用数组下标符号打印数组 number 的元素。使用 for 结构,假设已经声明了整型控制变量 I。打印每个数,精确到小数点后保留 1 位数。
- d) 用两条分离语句将数组 numbers 的开始地址赋值给指针变量 nPtr。
- e) 通过指针 nPtr 用指针/偏移量符号打印数组 numbers 的元素。
- f) 通过将数组名作为指针使用指针/偏移量符号打印数组 numbers 中的元素。
- g) 用带下标的指针 nPtr 打印数组 numbers 中的元素。
- h) 用数组下标符号、数组名的指针和指针/偏移量符号、nPtr 的指针下标符号和 nPtr 的指针/偏移量符号引用数组 numbers 的元素 4。
- i) 假设 nPtr 指向数组 number 开始位置, nPtr + 8 会引用哪个地址? 这个地址存放什么值?
- j) 假设 nPtr 指向 numbers[5], 执行 nPtr -= 4 之后, nPtr 会引用哪个地址? 这个地址存放什么值?

### 5.4 针对下列任务,各编写一条语句。假设已声明浮点数变量 number1 和 number2, number1 初始化为 7.3, 假设变量 ptr 为 char \* 类型, 数组 s1[100] 和 s2[100] 为 char 类型。

- a) 声明变量 fPtr 为指向 double 类型对象的指针。
- b) 将变量 number1 的地址赋值给指针变量 fPtr。
- c) 打印 fPtr 所指的对象的值。
- d) 指定 fPtr 所指的对象的值赋值给变量 number2。
- e) 打印 number2 的值。
- f) 打印 number1 的地址。
- g) 打印 fPtr 中存放的地址,打印的值是否与 number1 的地址相同?
- h) 将数组 s2 中存放的字符串复制到数组 s1。
- i) 比较 s1 中的字符串与 s2 中的字符串,并打印结果。
- j) 将 s2 中字符串的前 10 个字符添加到 s1 中的字符串。

k) 确定 s1 中的字符串的长度并打印结果。

l) 将 s2 中第一个标记的地址赋值给 ptr, s2 中的标记用逗号(,)分开。

#### 5.5 根据题目要求编写语句。

a) 为函数 exchange 编写函数首部,取双精度指针 x 和浮点数 y 两个指针作为参数,且不返回数值。

b) 写出 a) 中函数的函数原型。

c) 为函数 evaluate 编写函数首部,函数返回整数并且取整数 x 和函数 poly 的指针参数,函数 poly 取一个整数参数并返回一个整数。

d) 写出 c) 中函数的函数原型。

e) 显示用元音字符串“AEIOU”初始化字符数组 vowel 的两种不同方法。

#### 5.6 指出下列程序段中的错误。假设:

```
int * zPtr;           //zPtr will reference array z
int * aPtr = 0;
void * sPtr = 0;
int number, i;
int z[5] = {1,2,3,4,5};

sPtr = z;
```

a) ++zPtr;

b) //use pointer to get first value of array  
number = zPtr;

c) //assign array element 2(the value 3) to number  
number = \*zPtr[2];

d) //print entire array z  
for (i = 0; i <= 5; i++)  
cout << zPtr[i] << endl;

e) //assign the value pointed to by sPtr to number  
number = \*sPtr;

f) ++z;

g) char s[10];  
cout << strncpy(s, "hello", 5) << endl;

h) char s[12];  
strcpy(s, "Welcome Home");

i) if (strcmp(string1, string2))  
cout << "The strings are equal" << endl;

#### 5.7 执行下列语句时会打印什么结果(如果有)? 如果语句中有错,请指出并说明如何改正。假设下列变量声明:

```
char s1[50] = "jack", s2[50] = "jill", s3[50], *sPtr;
```

a) cout << strcpy(s3, s2) << endl;

b) cout << strcat(strcat(strcpy(s3, s1), "and"), s2)  
endl;

c) cout << strlen(s3) << endl;

## 自测题答案

5.1 a) 地址 b) 0, NULL 或地址 c) 0

5.2 a) 错误。地址操作符只能用于变量,不能用于常量、表达式或用存储类 `register` 声明的变量。

b) 错误。void 指针无法复引用,因为无法确切地知道要用多少内存字节复引用。

c) 错误。void 类型指针可以赋值给其他类型的指针,void 类型指针必须通过显式类型转换才可以赋给其他类型的指针。

5.3 a) `double numbers [SIZE] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};`

b) `double *nPtr;`

c) `cout << setiosflags (ios::fixed | ios::showpoint) << setprecision(1);`  
`for (i=0; i < SIZE; i++)`  
`cout << numbers[i] << ";`

d) `nPtr = numbers;`  
`nPtr = &numbers[0];`

e) `cout << setiosflags (ios::fixed | ios::showpoint) << setprecision(1);`  
`for (i=0; i < SIZE; i++)`  
`cout << *(nPtr+i) << ";`

f) `cout << setiosflags (ios::fixed | ios::showpoint) << setprecision(1);`  
`for (i=0; i < SIZE; i++)`  
`cout << *(numbers+i) << ";`

g) `cout << setiosflags (ios::fixed | ios::showpoint) << setprecision(1);`  
`for (i=0; i < SIZE; i++)`  
`cout << nPtr[i] << ";`

h) `numbers[3]`  
`*(numbers+3)`  
`nPtr[3]`  
`*(nPtr+3)`

i) 地址是  $10025000 + 8 * 4 = 1002532$ 。值为 8.8。

j) `numbers[5]` 的地址为  $1002500 + 5 * 4 = 1002520$ 。

`nPtr -= 4` 的地址为  $1002520 - 4 * 4 = 1002504$ 。

该地址的值为 1.1。

5.4 a) `double *fPtr;`

b) `fPtr = &number1;`

c) `cout << "The value of *fPtr is" << *fPtr << endl;`

d) `number2 = *fPtr;`

e) `cout << "The value of number2 is" << number2 << endl;`

f) `cout << "The address of number1 is" << &number1 << endl;`

g) `cout << "The address stored in fPtr is" << fPtr << endl;`

是的,其值相同。

h) `strcpy(s1,s2);`

i) `cout << "strcmp(s1,s2) = " << strcmp(s1,s2) << endl;`

j) `strncat(s1,s2,10);`

k) `cout << "strlen(s1) = " << strlen(s1) << endl;`

l) `ptr = strtok(s2,",");`

5.5 a) `void exchange(double *x, double *y)`

b) `void exchange(double *, double *);`

c) `int evaluate(int x, int(*poly)(int))`

d) `int evaluate(int, int(*) (int));`

e) `char vowel[] = "AEIOU";`

`char vowel[] = {'A','E','I','O','U','\0'};`

5.6 a) 错误:zPtr 没有被初始化。

改正:用 `zPtr = z;` 初始化 zPtr。

b) 错误:指针不被复引用。

改正:将该语句变成 `number = *zPtr;`

c) 错误;zPtr[2]不是指针,不能复引用。

改正:将 `zPtr[2]` 变为 `*zPtr[2]`

d) 错误:指针下标引用数组界限之外的数组元素。

改正:将 for 结构中的关系操作符变为“<”以避免指针下标引用数组界限之外的数组元素。

e) 错误:复引用 void 指针。

改正:要复引用指针,首先要将其转换为整型指针。将上述语句变为:

`number = *(int *)sPtr;`

f) 错误:试图用指针算法修改数组名。

改正:用指针变量代替数组名完成指针算法,或将数组名加上下标引用特定元素。

g) 错误:函数 `strncpy` 没有将空中止符写入组数 s,因为第 3 个参数等于字符串“hello”的长度。

改正:将 `strcpy` 的第 3 个参数变为 6 或对 `s[5]` 赋值 `'\0'`,确保在字符串后加上空中止符。

h) 错误:字符数组 s 不够大,不能存放空中止符。

改正:声明更多元素的数组。

i) 错误:函数 `strcmp` 在字符串相等时返回 0,因此 if 结构的条件为假,不执行输出语句。

改正:在 if 结构条件中将 `strcmp` 的结果与 0 比较。

5.7 a) jill

b) jack and jill

c) 8

d) 13

## 练习题

5.8 判断正误,如果有错,请说明原因。

- a) 比较指向不同数组的两个指针是没有意义的。
- b) 由于数组名是指向数组中第一个元素的指针,因此数组名可以和指针一样进行操作。

5.9 回答下列问题。假设无符号整数存放在 2 字节中,数组的开始内存地址为 1002500。

- a) 声明 5 个元素的 unsigned int 类型数组 values,并将元素初始化为 2 到 10 的偶数,假设已经将符号化常量 SIZE 定义为 5。
- b) 声明指针 vPtr,指向 unsigned int 类型的对象。
- c) 用数组下标符号打印数组 values 的元素。使用 for 结构,并假设已经声明整型控制变量 i。
- d) 用两个不同的语句将数组 values 的开始地址赋给针变量 vPtr。
- e) 用指针/偏移量符号打印数组 values 的元素。
- f) 随数组名用指针/偏移量符号将数组 values 的元素作为指针打印。
- g) 用带下标的数组指针打印数组 values 的元素。
- h) 随数组名用数组下标符号指针/偏移量符号将 values 的元素 5 作为指针、指针下标符号和指针/偏移量符号引用。
- i) vPtr + 3 引用什么地址? 该地址存放什么值?
- j) 假设 vPtr 指向 values[4],vPtr -= 4 引用什么地址,该地址存放什么值?

5.10 针对下列操作,编写执行指定任务。假设已声明整型变量 value1 和 value2,并且 value1 初始化为 20 000。

- a) 声明变量 lPtr 为 long 类型对象的指针。
- b) 将变量 value1 的地址赋给指针变量 lPtr。
- c) 打印 lPtr 所指的对象的值。
- d) 将 lPtr 所指对象值赋给变量 value2。
- e) 打印 value2 的值。
- f) 打印 value1 的地址。
- g) 打印 lPtr 中存放的地址,打印的值是否与 value 的地址相同?

5.11 根据题目要求编写语句。

- a) 为函数 zero 编写函数首部,取长整数数组参数 bigIntegers,不返回数值。
- b) 写出 a) 中函数的函数原型。
- c) 为函数 add1AndSum 编写函数首部,取整数数组参数 oneTooSmall 并返回一个整数值。
- d) 写出 c) 中函数的函数原型。

说明:练习题 5.12 ~ 5.15 有相当的难度。完成这些练习题后,即可轻松实现最常见的扑克牌游戏。

5.12 修改图 5.24 中的程序,令洗牌手向牌手发 5 张牌,然后编写完成下列任务的函数:

- a) 确定手中是否有一对牌。
- b) 确定手中是有对牌。

- c) 确定手中是否有 3 色同号牌(如 3 张 J)。  
 d) 确定手中是否有 4 色同号牌(如 4 张 K)。  
 e) 确定手中是否有同花(即 5 张牌花色相同)。  
 f) 确定手中是否有一条龙(即 5 张牌号连续)。
- 5.13 用练习题 5.12 开发的函数编写一个程序,发两手五张牌. 确定两手哪个更好。
- 5.14 修改练习题 5.13 中开发的程序,模拟发牌器。发牌器的五张牌是盖起来的,游戏者看不到。然后程序求值这手牌,根据牌的质量,抓一张、两张或三张,换掉原来手中不要的牌子。然后程序重新求值这手牌。(注意:这是个难题。)
- 5.15 修改练习题 5.14 开发的程序,命名其能自动处理发牌器中的牌,但游戏者可以确定自己手中要换的牌。然后程序求值这手牌,确定谁赢。用这个程序与计算机玩 20 次,看看是你赢还是计算机赢。再让你的朋友与计算机玩 20 次,看看谁赢的多。根据这些游戏的结果,完善扑克游戏程序(又是个难题)。再与计算机玩 20 次,修改后的程序是否更好玩?
- 5.16 在图 5.24 的洗牌和发牌程序中,我们故意用无效的洗牌算法,介绍引入无限延迟的可能性。在该练习题中,要生成高性能的洗牌算法以避免无限延迟。  
 按下面的做法修改图 5.24。初始化 deck 数组(如图 5.38)。修改 shuffle 函数,在数组中逐行逐列地循环,到达每个元素一次。每个元素与随机选择的数组元素进行交换。打印结果数组,确定是否洗好了牌(如图 5.39)。程序可能多次调用 shuffle 函数,才能保证洗好牌。

| 未洗牌的 deck 数组 |    |    |    |    |    |    |    |    |    |    |    |    |    |
|--------------|----|----|----|----|----|----|----|----|----|----|----|----|----|
|              | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 0            | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 1            | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 2            | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 3            | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |

图 5.38 未洗牌的 deck 数组

注意,尽管该练习题改进了洗牌算法,但洗牌算法仍然要从 deck 数组搜索第 1 张牌、第 2 张牌、第 3 张牌等等。更糟的是,即使在发牌算法找到并发出牌之后,该算法仍然会搜索其他牌。修改图 5.24 的程序,使发牌之后不再继续匹配这张牌,程序立即转入发下一张牌。

| 洗牌后的 deck 数组示例 |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------------|----|----|----|----|----|----|----|----|----|----|----|----|----|
|                | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 0              | 19 | 40 | 27 | 25 | 36 | 46 | 10 | 34 | 35 | 41 | 18 | 2  | 44 |
| 1              | 13 | 28 | 14 | 16 | 21 | 30 | 8  | 11 | 31 | 17 | 24 | 7  | 1  |
| 2              | 12 | 33 | 15 | 42 | 43 | 23 | 45 | 3  | 27 | 32 | 4  | 47 | 26 |
| 3              | 50 | 38 | 52 | 39 | 48 | 51 | 9  | 5  | 37 | 49 | 22 | 6  | 20 |

图 5.39 洗牌后的 deck 数组示例



- 5.17 (模拟龟兔赛跑)该练习再次讲述龟兔赛跑的故事。用随机数产生器开发模拟这个令人难忘的故事。

对手从 70 个的第 1 格起跑,每格表示跑道上的一個可能位置,终点线在第 70 格处。奖励第一个到达或者越过终点线的选手一个新鲜萝卜和莴苣。兔子要在山坡上睡一觉,因此可能失去冠军。

当时钟每秒钟滴答一次,程序应按下列规则调整动物的位置。

| 动物 | 运动类型 | 时间百分比 | 实际移动量   |
|----|------|-------|---------|
| 乌龟 | 快速爬行 | 50%   | 右移 3 格  |
|    | 滑倒   | 20%   | 左移 6 格  |
|    | 缓慢爬行 | 30%   | 右移 1 格  |
| 兔子 | 睡觉   | 20%   | 原地不动    |
|    | 大步跳  | 20%   | 右移 9 格  |
|    | 大步滑倒 | 10%   | 左移 12 格 |
|    | 小步跳  | 30%   | 右移 1 格  |
|    | 小步滑倒 | 20%   | 左移 2 格  |

用变量显示动物的位置(即位置号 1 到 70)。每个动物从位置 1 开始,如果动物跌到第 1 格以外,则移回第 1 格。

产生随机整数  $i$  ( $1 \leq i \leq 10$ ),以得到上表中的百分比。对于乌龟,  $1 \leq i \leq 5$  时快走,  $6 \leq i \leq 7$  滑倒,  $8 \leq i \leq 10$  时慢走,兔子也用相似的方法。

起跑时,打印

BANG!!!!

AND THEY'RE OFF !!!!!

时钟每次滴答一下(即每个重复循环),打印第 70 格位置的一条线,显示乌龟的位置 T 和兔子的位置 H。如果两者占用一格,则乌龟会咬兔子,程序从该位置开始打印 ouch!!!。除了 T、H 和 OUCH!!! 以外的其他打印位置都是空的。

打印每一行之后,测试某个动物是否超过了第 70 格。如果是,则打印获胜者,停止模拟。如果乌龟赢,则可以同情弱者,让乌龟赢,或者打印 "It's a tie"。如果两者都没有赢,则再次循环,模拟下一个时钟滴答。准备运行程序时,让一组啦啦队看比赛,你会发现观众非常投入。

### 特色部分:建立自己的计算机

对于高级语言编程,下面几个问题属于题外话。我们打开计算机,看看其内部结构。我们介绍机器语言编程和编写几个机器语言程序。这些知识很有价值,我们要建立自己的计算机(通过软件模拟技术),用于执行我们的机器语言程序。

- 5.18 (机器语言程序)下面要建立一个 simpletron 计算机。顾名思义,这是个简单机器,但是稍后你会发现它具有强大的功能。Simpletron 只能运行用 Simpletron Machine Language (SML 机器语言)写成的程序。

Simpletron 包含一个累加器(特殊寄存器),存放 simpletron 用于计算和各种处理的信息,Simpletron 处理的所有信息都为单位“字”以处理。字是带符号的四位十进制数,如

+3364, -1293, +0007, -0001 等等。Simpletron 带有 100 个字的内存, 这些字用其内存单元号 00, 01, ..., 99 引用。

运行 SML 程序之前, 要先把程序载入内存。每个 SML 程序中的第一条指令(或语句)一般放在内存单元 00 处, 模拟器会从该地址开始执行。

用 SML 编写的每条指令都占用 simpletron 内存中的一个字。(因此, 指令是带符号的 4 位十进制数)。我们应假设 SML 指令的符号总是正号, 但数据字的符号可正可负。Simpletron 内存中的每个单元可以包含一条指令, 程序使用的数据值未用(未定义)内存区。每个 SML 指令的前两位是操作码, 指定要进行的操作。图 5.37 显示了 SML 操作码。

| 操作码                        | 意义                                     |
|----------------------------|----------------------------------------|
| 输入/输出操作:                   |                                        |
| const int READ = 10;       | 从键盘读一个字到特定内存单元                         |
| const int WRITE = 11;      | 从特定内存单元写一个字到屏幕                         |
| 装入/保存操作:                   |                                        |
| const int LOAD = 20;       | 从特定内存单元将字装入累加器                         |
| const int STORE = 21;      | 将累加器中的字存放到特定内存单元                       |
| 算术运算:                      |                                        |
| const int ADD = 30;        | 将特定内存单元中的字加到累加器中的字(结果保留在累加器中)          |
| const int SUBTRACT = 31;   | 将特定内存单元中的字减去累加器中的字(结果保留在累加器中)          |
| const int DIVIDE = 32;     | 将特定内存单元中的字除以累加器中的字(结果保留在累加器中)          |
| const int MULTIPLY = 33;   | 将特定内存单元中的字乘以累加器中的字(结果保留在累加器中)转移到特定内存单元 |
| 控制转移操作:                    |                                        |
| const int BRANCH = 40;     | 转移到特定内存单元                              |
| const int BRANCHNEG = 41;  | 如果累加器为负值, 则转移到特定内存单元                   |
| const int BRANCHZERO = 42; | 如果累加器为 0, 则转移到特定内存单元                   |
| const int HUT = 43;        | 停止, 程序已完成任务                            |

图 5.40 SML 机器语言操作码

SML 指令的最后两位是操作数, 也就是要操作的字的内存单元地址。

下面要思考几个简单的 SML 程序。第一个 SML 程序(示例①)从键盘读取两个数, 并计算和打印这两个数的和。指令 +1007 从键盘读取第一个数并将其放在内存单元 07 (初始化为 0)中, 然后指令 +1008 读取下一个数并将其放在内存单元 08 中。装入指令 +2007 将第一个数放(复制)到累加器中, 加法指令 +3008 将第二个数与累加器中的数相加。所有 SML 算法指令都把结果留在累加器中。保存指令 +2109 将结果放(复制)回内存单元 09, 写指令 +1109 取得并打印这个结果(带符号的四位十进制数), 停止指令 +4300 中止程序执行。

| 示例①位置 | 数字    | 指令           |
|-------|-------|--------------|
| 00    | +1007 | (Read A)     |
| 01    | +1008 | (Read B)     |
| 02    | +2007 | (Load A)     |
| 03    | +3008 | (Add B)      |
| 04    | +2109 | (Store C)    |
| 05    | +1109 | (Write C)    |
| 06    | +4300 | (Halt)       |
| 07    | +0000 | (Variable A) |
| 08    | +0000 | (Variable B) |
| 09    | +0000 | (Result C)   |

示例②中的 SML 程序从键盘读取两个数,并确定和打印其中较大值。注意用指令 +4107 作为条件控制转移,使之类似于C++ 中的 if 语句。

| 示例②位置 | 数字    | 指令                      |
|-------|-------|-------------------------|
| 00    | +1009 | (Read A)                |
| 01    | +1010 | (Read B)                |
| 02    | +2009 | (Load A)                |
| 03    | +3110 | (Subtract B)            |
| 04    | +4107 | (Branch negative to 07) |
| 05    | +1109 | (Write A)               |
| 06    | +4300 | (Halt)                  |
| 07    | +1110 | (Write B)               |
| 08    | +4300 | (Halt)                  |
| 09    | +0000 | (Variable A)            |
| 10    | +0000 | (Variable B)            |

现在编写 SML 程序,完成下列任务:

- 用标记控制循环读取正数,并计算和打印它们的和。当遇到负值时,停止输入。
- 用计数器控制循环读取 7 个数,其中包括正数和负数,计算并打印它们的平均值。
- 读取一系列数,并确定和打印最大的数。第一个读取的数表示要处理多少个数。

- 5.19 (计算机模拟程序)最初看起来似乎有点不可理解,但是在这个问题中,要建立自己的计算机。不是指把计算机的硬件连接起来,而是指用软件基础的模拟器的强大技术建立一个 simpletron 软件模型。大家不会失望的,该模拟器可以将读者正在使用的计算机变成 simpletron 并能实际运行、测试和调试练习题 5.18 中编写的 SML 程序。

运行 simpletron 模拟器时,打印

```

***Welcome to Simpletron! ***

*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark(?). ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***

```

```
***your program.***
```

用 100 个元素的单下标数组 `memory` 模拟 `simpletron` 内存。然后假设正在运行这个模拟程序,并检查一下练习题 5.18 中示例②的对话:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

***Program loading completed***
***Program execution begins***
```

这时 SML 程序已经载入(装入)数组 `memory` 中, `simpletron` 开始执行 SML 程序。程序从内存单元 00 的指令开始执行,和 C++ 一样,按顺序继续执行,但可以通过定向控制转移入程序的其他部分。

用变量 `accumulator` 表示累加寄存器。用变量 `counter` 跟踪内存中包含所执行指令的内存单元。用变量 `operationcode` 表示当前正在进行的操作,即指令字的左边两位。用变量 `operand` 表示操作当前指令的内存单元即指令字的右边两位。因此, `operand` 是当前正在执行的指令的最右边两位数字。不要直接从内存中执行指令,而是将下一个要执行的指令从内存中转到变量 `instructionRegister` 中。然后选取左边两位,放进 `operationCode` 中,选取右边两位,放进 `operand` 中。当 `Simpletron` 开始执行时,所有特殊寄存器都初始化为 0。

下面看一下第一个 SML 指令(内存地址 00 的指令 +1009)的执行过程。这被称为指令执行循环。

`Counter` 指出下一个要执行指令的内存单元。我们用 C++ 语句

```
InstructionRegister = memory[counter];
```

从 `memory` 中取得该地址。

语句

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

从指令寄存器选取操作码和操作数。现在 `Simpletron` 必须确定操作码是读(不是写、装入等)。Switch 结构区分 SML 的 12 种操作。

在 `switch` 结构中,各种 SML 操作指令的模拟如下(其他留给读者练习):

```
读          cin  >> memory[ operand ];
载入        accumulator = memory[ operand ];
```

```

加          accumulator += memory[ operand ];
转移        稍后将介绍转移
停止        这条指令打印下列信息
            *** Simpletron execution terminated ***

```

然后打印每个寄存器的名称与内容,是指内存的完整内容。这种打印输出通常被称为计算机转储(计算机转储不是指旧计算机被放置的地方)。为了帮助编制转储功能,图 5.41 显示了一个示例转储格式。注意执行 simpletron 程序之后,计算机转储显示指令实际值和中止执行时的数据值。

输出结果:

```

REGISTERS:
accumulator      +0000
counter          00
instructionRegister +0000
operationCode     00
operand          00

MEMORY:

```

|    | 0     | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     | 9     |
|----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 10 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 20 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 30 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 40 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 50 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 60 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 70 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 80 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 90 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |

图 5.41 转储示例

下面执行我们程序的第一条指令,内存单元 00 中的 +1009。如前所述,用 C++ 的 switch 语句

```
cin >> memory[operand];
```

模拟这个过程。

执行 cin 之前,屏幕上显示一个问号(?),提示用户输入。Simpletron 等待用户输入一个值并按 Return 键。然后这个值读取到内存单元 09。

这时,第一个指令的模拟已经完成。余下的就准备让 simpletron 执行下一条指令。由于刚才执行的指令不是控制转移,因此只需增加指令计数器,如下所示

```
++counter;
```

至此已完成了第一条指令的模拟。整个过程(指令执行循环)重新开始,读取下一条要执行的指令。

现在我们需要考虑一下如何模拟转移指令(控制转移),我们需要的只是调整指令计数器的值。因此,无条件转移指令(40)可以用 switch 语句模拟如下

```
counter = operand;
```

条件指令“累加器为0”则转移)模拟如下

```
if ( accumulator == 0)
    counter = operand;
```

这时就可以实现 Simpletron 模拟程序,并且可以运行练习题 5.18 中编写的每一个 SML 程序。可以在 SML 中增加其他特性,并在模拟程序中提供这些特性。

模拟程序应检查各种类型的错误。例如,程序装入期间,用户输入 Simpletron 程序 memory 中的每个数应在 -9999 ~ +9999 范围之内。模拟程序应该用 while 循环测试输入的每个数是否在这个范围中,如果不在,则提示用户重新输入,直到用户输入正确的数。

在执行过程中,Simpletron 模拟程序应检查各种严重错误,例如,除数为 0、执行无效操作码、累加器溢出(即算术操作的结果不在 -9999 ~ +9999 之间)等等。这种严重错误被称为致命错误。发生致命错误时,模拟程序应打印下列错误消息

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

并按照前面介绍的格式打印完整的计算机转储,帮助用户找出程序中的错误。

### 更多指针练习题

5.20 修改图 5.24 中的洗牌与发牌程序,使洗牌与发牌操作由同一个函数 ShuffleAndDeal 完成。这个函数应包含类似于图 5.24 中的函数的嵌套结构。

5.21 指出下列程序的用途。

```
1 //ex05_21.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 void mystery1( char *, const char * );
9
10 int main()
11 {
12     char string1[ 80 ], string2[ 80 ];
13
14     cout << "Enter two strings: ";
15     cin >> string1 >> string2;
16     mystery1( string1, string2 );
17     cout << string1 << endl;
18     return 0;
19 }
20
21 void mystery1( char *s1, const char *s2 )
22 {
23     while ( *s1 != '\0' )
```

```
24     ++s1;
25
26     for ( ; *s1 = *s2; s1 ++, s2 ++ )
27         ;    //empty statement
28 }
```

### 5.22 指出下列程序的用途。

```
1  //ex05_22.cpp
2  #include <iostream>
3
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int mystery2( const char * );
9
10 int main()
11 {
12     char string[ 80 ];
13
14     cout << "Enter a string: ";
15     cin >> string;
16     cout << mystery2( string ) << endl;
17     return 0;
18 }
19
20 int mystery2( const char *s )
21 {
22     int x;
23
24     for ( x = 0; *s != '\0'; s ++ )
25         ++x;
26
27     return x;
28 }
```

### 5.23 指出下列程序中的错误,并说明如何改正:

- a) `int *number;`  
    `<<number <<endl;`
- b) `double *realPtr;long *integerPtr;`  
    `integerPtr=realPtr;`
- c) `int *x,y;`  
    `x=y;`
- d) `char s[] = "this is a character array";`  
    `for ( ; *s!= '\0'; s ++ )`  
        `cout << *s << " ;`
- e) `short *numPtr, result;`  
    `void *genericPtr=numPtr;`

```

    result = *genericPtr + 7;
f) double x = 19.34;
    double xPtr = &x;
    cout << xPtr << endl;
g) char *s;
    cout << s << endl;

```

5.24 (快速排序)在第4章的例子和练习题中,我们介绍了冒泡排序、桶排序和选择排序的排序技巧。现在要介绍名为快速排序的递归排序方法。单下标数组的基本算法如下:

a) 分区步骤:取未排序数组的第一个元素,确定其在排序数组中的最终位置,即该元素左边的所有值小于该元素,该元素右边的所有值大于该元素。这个就确定了一个元素的位置,有了两个未排序子数组。

b) 递归步骤:对每一个未排序子数组执行第一个步骤。

每次对未排序数组执行第一个步骤时,另一个元素被放在排序数组的最终位置上,这样就生成两个未排序子数组。当子数组只包含一个元素时,它必须排序,该元素在最终位置上。

基本算法似乎很简单,但如何确定每个数组的第一个元素在排序数组中的最终位置呢?例如,考虑下列数值

37 2 6 4 89 8 10 12 68 45

黑体元素是分区元素,要放在排序数组中的最终位置。

a) 从数组最右边的元素开始,将每个元素与37进行比较,直到找出小于37的元素,然后将这个元素与37交换。第一个小于37的元素是12,因此将12与37交换。新数组为

12 2 6 4 89 8 10 37 68 45

元素12用斜体显示,表示刚刚与37交换。

b) 从这个数组左边12以后的元素开始,将每个元素与37比较,直到找到大于37的元素,然后将这个元素与37交换。第一个大于37的元素是89,将89与37交换。新数组如下:

12 2 6 4 37 8 10 89 68 45

c) 从数组89以前的元素向右开始,将每个元素与37比较,直到找出小于37的元素,然后将这个元素与37交换。第一个小于37的元素是10,将10与37交换。新数组如下为

12 2 6 4 10 8 37 89 68 45

d) 从数组左边10以后的元素开始,将每个元素与37开始,直到找出大于37的元素,然后将这个元素与37交换。由于没有比37更大的元素,因此37已经放在排序数组中的最终位置。

对上述数组分区后,有两个未排序子数组。小于37的未排序的子数组包含12,2,6,4,10和8。大于37的未排序子数组包含89,68和45。对这两个未排序子数组进行像原数组一样的处理。

根据上述介绍,编写一个递归函数 quicksort,排序单下标整型数组。函数应接收一个整型数组,作为开始下标和结束下标参数。函数 quicksort 调用函数 partition 进行



分区。

5.25 (走迷宫)下列#和.组成的网络是表示迷宫的双下标数组:

```

# # # # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # . #
# # # . # . . . # . #
# . . . . # # # . # .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # # #

```

在上述双下标数组中,#表示迷宫的墙,圆点表示可以走的路线,只能在数组中包含圆点的地方移动。

有一个简单算法保证找到迷宫的出口(如果有出口)。如果没有出口,则会回到起始位置。将右手放在右边的墙上开始前进,手不离开墙,如果迷宫向右转,手也跟着向右前进。只要手不离开墙,最终总能走到出口。当然,可能还有更短的路径,但是按照上述途径做可以保证找到出口。

编写走迷宫的递归函数 `mazeTraverse`。函数接收表示迷宫和迷宫起始位置的  $12 \times 12$  字符数组作为参数。`mazeTraverse` 在试图寻找迷宫的出口时,应将字符 `x` 放在沿途的每一格。每次移动之后,函数应显示迷宫,让用户可以看到迷宫的走法。

- 5.26 (随机产生迷宫)编写一个 `mazeGenerator`,接收  $12 \times 12$  字符数组作为参数并随机产生迷宫。函数还应提供迷宫的开始和结束位置。试用随机产生迷宫测试练习 5.25 所写的函数 `mazeTraverse`。
- 5.27 (任何大小的迷宫)将练习题 5.25 和练习题 5.26 的函数 `mazeTraverse` 与 `mazeGenerator` 一般化,用于处理任何大小的迷宫。
- 5.28 (函数指针数组)将图 4.23 中的程序改写成使用菜单驱动界面。程序为用户提供了 5 个选项,如下所示(应在屏幕上显示出来):

```

Enter a choice:
0  Print the array of grades
1  Find the minimum grade
2  Find the maximum grade
3  Print the average on all tests for each student
4  End program

```

使用函数指针数组的一个限制是所有指针必须是同一类型。指针必须指向接收相同类型参数和返回相同类型数值的函数。因为这个原因,图 4.23 中的函数必须修改成每次返回相同类型和接收相同类型的参数。将函数 `minimum` 和 `maximum` 修改成打印最小值与最大值,不返回任何内容。对选项 3,将图 4.23 的函数 `average` 修改成输出每个学生(而不是特定学生)的平均成绩。函数 `average` 与 `printArray`, `minimum` 和 `maximum` 接收相同类型参数且不返回任何内容。将 4 个函数的指针存放在数组 `process-`

Grades 中,并用用户选择的选项作为调用每个函数的数组下标。

5.29 (修改 Simpletron 模拟器) 练习题 5.19 中编写了计算机的软件模拟,执行用 Simpletron Machine Language(SML)编写的程序。在这个练习中,要对 Simpletron 模拟器进行修改和补充。练习题 15.26 和练习题 15.27 中建立一个编译器,将编写程序的语言由高级编程语言(由 BASIC 转变而来)转换为 SML 语言编写。要执行编译器产生的程序,需要进行下列修改和补充。请注意,一些修改可能会发生冲突,因此必须分开操作。

- a) 将 Simpletron 模拟器的内存扩展为包含 1000 个内存单元,使 Simpletron 模拟器能处理更大的程序。
- b) 让模拟器执行求模计算,这要求增加 SML 指令。
- c) 让模拟器执行求幂计算,这要求增加 SML 指令。
- d) 将 Simpletron 模拟器修改成用十六进制值而不是用整数值表示 SML 指令。
- e) 将 Simpletron 模拟器修改成允许输出换行符。这要求增加 SML 指令。
- f) 将 Simpletron 模拟器修改成不仅能处理整数值,而且能处理浮点数。
- g) 将 Simpletron 模拟器修改成处理字符串输入。提示:每个 Simpletron 字可以分为两组,每一组分别放两位整数。每个两位整数表示一个字符的 ASCII 十进制对应值。增加在特定的 Simpletron 内存单元开始输入和存放字符串的机器语言指令。该内存单元的字的前半部分是字符串中的字符数(即字符串的长度)。后半部分包含一个用两个十进制位所表示的 ASCII 值并赋给半个字。
- h) 将模拟器修改成处理(按 g)中的格式存放的)字符串输出。提示:增加在特定 Simpletron 内存单元开始打印字符串的机器语言指令。该内存单元的字的前半部分是字符串中的字符数(即字符串的长度)。后半部分包含一个用两个十进制位表示的 ASCII 字符。机器语言指令通过将两位数转换成相应字符来打印字符串来检查字符串的长度并打印字符串。
- i) 将模拟器修改为包含指令 SML\_DEBUG,在每一个指令执行完之后,打印内存转储。SML\_DEBUG 是一个 44 位的操作编码。+4401 打开调试模式,+4400 断开调试模式。

5.30 指出下列程序的用途。

```

1 //ex05_30.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 bool mystery3( const char *, const char * );
9
10 int main()
11 {
12     char string1[ 80 ], string2[ 80 ];
13
```

```
14     cout << "Enter two strings: ";
15     cin >> string1 >> string2;
16     cout << "The result is "
17         << mystery3( string1, string2 ) << endl;
18
19     return 0;
20 }
21
22 bool mystery3( const char *s1, const char *s2 )
23 {
24     for ( ; *s1 != '\0' && *s2 != '\0'; s1 ++, s2 ++ )
25
26         if ( *s1 != *s2 )
27             return false;
28
29     return true;
30 }
```

### 字符串操作练习

- 5.31 编写一个程序,用函数 `strcmp` 比较用户输入的两个字符串。该程序应说明第一个字符串是小于、等于或大于第二个字符串。
- 5.32 编写一个程序,用函数 `strncmp` 比较用户输入的两个字符串。该程序应说明第一个字符串是小于、等于或大于第二个字符串。
- 5.33 编写一个程序,用随机数产生器建立生成语句。该程序应用 4 个 `char` 类型的指针数组,它们分别是 `article`, `noun`, `verb` 和 `preposition`。该程序应按下列顺序从 4 个数组中随机选取一个元素生成一个语句: `article`, `noun`, `verb`, `preposition`, `article` 和 `noun`。当选取每个单词时,应该特别注意上述单词组成的数组是否足够大,要以放下整个语句。单词之间就用空格分开。输出最后的语句时,应以大写字母开头,以圆点结尾,程序应生成 20 个语句。
- 语句填充如下: `article` 数组应包含冠词:“the”,“a”,“one”,“some”和“any”; `noun` 数组包含名词“boy”,“girl”,“dog”,“town”和“car”; `verb` 数组应包含动词:“drove”,“jumped”,“ran”,“walked”和“skipped”; `preposition` 数组应包含介词:“to”,“from”,“over”,“under”和“on”。
- 编写并执行上述程序之后,将程序修改成由几个句子组成的短故事(可以任意编写一篇学期报告)。
- 5.34 (5 行打油诗)5 行打油诗由 5 句话组成,第一行、第二行和第 5 行压韵;第 3 行与第 4 行压韵的幽默诗句。我们用练习题 5.33 中学习到的方法编写一个随机产生打油诗的 C++ 程序。要加工该程序以生成好的打油诗是颇有难度,但这个工作非常具有挑战性。
- 5.35 编写一个将英语短语编成 `pig latin` 的程序, `pig latin` 是一种故意打乱单词字母顺序的娱乐形式。形成 `pig latin` 有很多不同的方法。下面是一个简单的 `pig latin` 算法。
- 要将英语短语编成 `pig latin`,用函数 `strtok` 将短语标记化为各个单词。要每个英语单词

编译成 pig latin, 将英语单词的第一个字母放到该单词的末尾, 并加上字母“ay”, 如同“jump”变成“umpjay”, “the”变成“hetay”, “computer”变成“omputercay”。单词之间的空格保持不变。假设英语单词之间用空格分开, 没有标点符号且每个单词至少有两个字母组成。函数 printlatinword 就应显示每个单词。提示: 每次调用 strtok 并找到一个标记时, 将标记指针传递给函数 printlatinword, 并打印 pig latin 单词。

- 5.36 编写一个程序, 以(555)555-5555 的形式输入电话号码字符串。程序应用函数 strtok 取得区号标记, 电话号码的前3位作为一个标记, 后4位作为一个标记。电话号码的7位数就被连接成一个字符串。程序将区号字符串变为 int 类型, 将电话号码字符串变为 long 类型, 并打印电话号码和区号。
- 5.37 编写一个程序, 输入一行文本, 用函数 strtok 标记化该文本, 并以相反顺序输出标记。
- 5.38 用我们在 5.12.2 节介绍过的字符串比较函数和第4章介绍的数组排序技术编写一个程序, 按字母列出字符串清单。用10个或15个城市名作为程序数据。
- 5.39 对图 5.29 中的字符串复制和字符串连接函数编写两个版本, 第一个版本使用数组下标, 第二个版本使用指针与指针算法。
- 5.40 对图 5.29 的字符串比较函数编写两个版本, 第一个版本用数组下标, 第二个版本用指针与指针算法。
- 5.41 对图 5.29 的字符串 strlen 函数编写两个版本, 第一个数组下标, 第二个用指针与指针算法。

### 特色部分: 高级字符串操作练习

上述练习题是本书的关键部分, 用于测试读者对基本的字符串操作要领的理解。这一部分将介绍一组中高级字符串操作练习。读者会发现这些问题具有一定的难度, 但是却很有意思。这些问题的难度不一, 有一些需要花费一个或两个小时来编写和实现。有些需要两至三周来学习和实践, 有些是较难的小组项目。

- 5.42 (文本分析) 利用计算机的字符串操作功能可以用非常有趣的方法分析大作家的写作方法。很多人将注意力集中到莎士比亚是否真有其人, 一些学者确信, 有重要证据表明, 莎士比亚实际上是 Christopher Marlowe 或者其他大作家的化名。研究人员通过使用计算机寻找这两位作家在写作中的相似性。本练习题检验3种用计算机来分析文本的方法。

- a) 编写一个程序, 从键盘读取几行文本, 并打印一个表格, 显示文本中字母的出现次数。例如, 短语

To be, or not to be; that is the question;

包含一个 a, 两个 b, 不包含 c, 等等。

- b) 编写一个程序, 从键盘读取几行文本, 并打印一个表格, 显示文本中单字符单词、双字符单词、三字符单词等的出现次数。例如, 语句

Whether 'tis nobler in the mind to suffer

中包含

| 字长 | 出现次数      |
|----|-----------|
| 1  | 0         |
| 2  | 2         |
| 3  | 1         |
| 4  | 2(包括 tis) |
| 5  | 0         |
| 6  | 2         |
| 7  | 1         |

c) 编写一个程序,从键盘读取几行文本,并打印一个表格,显示文本中每个单词的出现次数。程序的第一个版本就在表中按文本中出现的顺序列出单词。例如,语句

```
To be, or not to be ;that is the question;
Whether 'tis nobler in the mind to suffer
```

d) 包含三个“to”,两个“be”,一个“or”,等等。然后按字母顺序列出更有趣(也更有用)的打印输出。

5.43 (字处理)字处理系统的一个重要功能是输入对齐,将单词与页面的左右边界对齐,从而产生漂亮的文档,看起来就像是经过排版的,而不是直接输入的。计算机系统在行中的单词之间插入空格而让最右边的单词与右边界对齐。

编写一个程序,取几行文本,并按对齐的格式打印这些文本。假设文本在 8.5 英寸宽的纸上打印,左右边界留下 1 英寸的边距。假设计算机在水平方向每英寸打印 10 个字符,因此可以打印 6.5 英寸的文本,或者是每行打印 65 个字符。

5.44 (按不同格式打印日期)在商务信函中,日期可以按几种不同的格式打印,两种比较常用的格式如下

07/21/1955 和 July 21,1955

编写一个程序,读取第一种日期格式的日期并以第二种日期格式打印日期。

5.45 (支票保护)计算机经常被就用在支票写入系统,例如:工资与账号支付应用等。许多怪事经常出现,如每周工资支票上错误地多写入 1 百万美元。由于人为的和机器的错误,使支票写入系统写入不正常的数值。系统设计人员在系统中建立控制,防止发生这种错误支票。有些人故意改变支票金额,想窃取钱财,这是一个非常严重的问题。为了防止篡改支票金额,大多数计算机的支票写入系统采用了支票保护技术。计算机打印的支票包含固定的可以打印金额的空间。假设支票中用 8 个空格填写每周工资,如果金额太大,则 8 个空格都会填满,例如:

1 230.60                   (支票金额)

12345678                   (位置号)

另一方面,如果金额少于 1 000 美元,则保留某些空格。例如:

99.87

12345678

有 3 个空格。如果打印支票时留下空格,则很容易被入篡改支票金额。要防止有人更

改支票金额,许多支票写入系统插入如下星号:

\*\*\*99.87

12345678

编写一个程序,输入支票上要打印的美元数,然后用支票保护格式打印金额,必要时加上星号,假设用九个空格打印金额。

- 5.46 (写入支票金额的大写形式)继续上面的例子,设计支票写入系统以防止改变支票金额非常重要,一个常用的安全方法就是填写支票金额的大写形式。假如支票的数字被改,但因大写金额很难篡改而无效。

许多计算机支票写入系统不写出支票大写金额,主要原因可能是商用应用程序使用的大多数高级语言没有足够的字符串操作特性,另一个原因则是编写大写金额的程序比较复杂。编写一个C++程序,输入数字金额。例如,金额112.43写为

ONE HUNDRED TWELVE AND 43 /100

- 5.47 (莫尔斯码)最著名的编码机制大概要算莫尔斯码了,这是塞缪尔·莫尔斯在1932年发明的。莫尔斯码用一组点和线来表示字母、数字和一些特殊符号(如圆点、逗号、分号等)。在面向声音的系统中,点表示短音,线表示长音。点线还表示用于面向光的系统和面向信号标志系统。

单词之间用空格分开,相当简单,无需使用点和线。在面向语音的系统中,空格是通过短时间不传输声音来表示的。图5.42所示的是莫尔斯码的国际化版本。

| 字符 | 代码      | 字符 | 代码        |
|----|---------|----|-----------|
| A  | . -     | T  | -         |
| B  | - ...   | U  | .. -      |
| C  | - . - . | V  | ... -     |
| D  | - . .   | W  | . - -     |
| E  | .       | X  | - . . -   |
| F  | .. - .  | Y  | - . - -   |
| G  | - - .   | Z  | - - . .   |
| H  | ....    | 数字 |           |
| I  | ..      | 1  | . - - - - |
| J  | . - - - | 2  | .. - - -  |
| K  | - . -   | 3  | ... - -   |
| L  | . - . . | 4  | .... -    |
| M  | - -     | 5  | .....     |
| N  | - .     | 6  | - ....    |
| O  | - - -   | 7  | - . - . . |
| P  | . - - . | 8  | - - - . . |
| Q  | - - . - | 9  | - - - - . |
| R  | . - .   | 0  | - - - - - |
| S  | ...     |    |           |

图5.42 莫尔斯码的国际化版本

编写一个程序,读取一句英语短语,并将其编写成莫尔斯码,再用一个程序将莫尔斯码还原为英语。莫尔斯码编码字母之间有一个空格,莫尔斯码编码单词之间有 3 个空格。

- 5.48 (公制换算程序)编写一个程序,帮助用户进行公制换算。程序允许用户指定单位字符串(如 centimeters, liters, grams 等表示公制, inches, quarts, pounds 等表示英制),并回答下列简单问题

“How many inches are in 2 meters?”

“How many liters are in 10 quarts?”

程序应识别出无效转换,例如,语句

“How many feet in 5 kilograms?”

是无意义的,因为“feet”是长度,而“kilograms”是重量单位。

### 复杂字符串操作练习

- 5.49 (纵横填字谜产生器)大多数人都玩过纵横填字谜游戏,但很少有人建立过纵横填字谜。建立一个填字谜程序具有一定难度,是个复杂字符串操作项目,需要有坚实的基础并付出相当多努力。程序员编写最简单的纵横填字谜程序,也要解决很多问题。例如,如何在计算机中表示纵横填字谜的网格?是用一系列字符串,还是用双下标数组?程序员需要程序直接引用的单词源(即计算机专业字典)。这些单词是以什么形式存放,以便实现程序所需要的复杂操作?有的读者还想建立纵横填字谜的“线索”,为解谜者提示每行每列要打印的单词。打印一个空的纵横填字谜也不是件容易的事。

## 第6章 类和数据抽象(一)

### 学习目标

- 理解封装与数据隐藏的软件工程概念
- 理解数据抽象和抽象数据类型(ADT)概念
- 会创建C++ ADT(即类)
- 理解如何创建、使用和删除类对象
- 能够控制对象数据成员和成员函数的访问
- 开始认识到“面向对象”的价值

### 6.1 简介

本章将介绍C++中的面向对象。为什么把C++中的面向对象编程延到第6章介绍呢?因为我们将要建立的对象由各个结构化程序组件构成,所以需要事先了解结构化程序的基础知识。

第1章到第5章结尾的“对象思想”部分,已经介绍了C++中面向对象编程的基本概念(即对象思想)和术语(即对象语言)。在这些特色部分中,我们还介绍了面向对象设计(OOD)的方法:分析典型问题语句,要求建立一个系统(电梯模拟程序),确定系统中需要实现的类,确定这些类对象的属性,确定这些类对象的行为,并指定对象之间如何通过交互来完成系统的总体目标。

下面简单回顾一下面向对象的一些关键概念和术语。面向对象编程(OOP)将数据(属性)和函数(行为)封装到称为类的软件包中:类的数据和函数紧密地联系在一起,类就像蓝图。建筑人员可以通过蓝图建造房子;程序员则通过类生成对象。一个蓝图可以反复使用多次,建造很多房子;一个类可以反复使用,生成多个同类的对象。类具有信息隐藏的属性。这就意味着尽管类对象只知道如何通过定义良好的接口与其他类对象通信,但是类通常不知道其他类的实现方法,即实现细节被隐藏在类中。如同我们在不知道发动机、传递系统和燃料系统的内部工作原理的情况下,仍能熟练驾驶汽车。由此可以看出信息隐藏对于良好的软件工程多么重要。

在C语言和其他过程性编程语言中,编程总是面向操作,然而在C++中编程是面向对象的。在C语言中,编程的单位是函数,而在C++中,编程的单位是类,面向对象最终要通过类实例化(即生成)。

C语言程序员把重点放在函数的编写上。完成某个任务的一组操作便构成了函数,多个函数的组合便构成程序。数据在C语言中的确很重要,但数据只存在于支持完成函数的初级操作中。系统指定中的动词说明有助于C语言程序员确定用于实现系统的一组函数集。



C++ 语言程序员则将重点放在生成类的用户自定义类型。类也称为程序员自定义型。每一个类包含数据和操作数据的函数集。类的数据部分称为数据成员;类的函数部分称为成员函数(或在其他面向对象语言中称为方法)。内部类型的实例(如 int)称为变量;用户自定义类型的实例(如类)称为对象。在C++中,变量和对象常常互换使用。C++的重点不是函数而是类。系统指定中的名词说明有助于C++程序员确定实现系统所需的用来生成对象的一组类。

C++中的类从C语言中的 struct 概念自然演变而来。介绍C++类的开发之前,我们先介绍结构,在结构基础上建立用户自定义类型。先介绍类的不足,将有助于说明类的概念。

## 6.2 结构定义

结构是混合数据类型,由其中包含 struct 的其他类型元素构成。结构定义

```
struct Time{
    int hour; //0-23
    int minute; //0-59
    int second; //0-59
}
```

中,关键字 struct 引入结构定义。标识符 Time 是结构标记,命名结构定义并声明该结构类型的变量。在这个例子中,新类型名为 Time。结构定义花括号中声明的名称是结构的成员。同一结构中成员名称必须是惟一的,但是两个不同的结构可以包含同一名称的成员,而不会发生冲突。每个结构定义必须以分号结尾。上述解释对稍后介绍的类也有效;在C++中,结构和类是非常相似的。

Time 的定义包含 3 个 int 类型的成员:hour, minute 和 second。结构成员可以是任意类型,一个结构可以包含很多不同类型的成员。但是结构不能包含自身的实例。例如,Time 类型成员不能在 Time 的结构定义中声明。但是,可以包含另一个 Time 结构的指针。包含同一结构类型指针的结构称为自引用结构。自引用结构用于形成链接数据结构,例如:链表、队列、堆栈和树等(参见第 15 章)。

上述结构定义在内存中不保留任何空间,而是生成一个用于声明变量的新数据类型。声明

```
Time timeObject, timeArray[10], *timePtr,
    &timeRef = timeObject;
```

将 timeObject 声明为 Time 类型的变量, timeArray 是 Time 类型的 10 个元素的数组, timePtr 是 Time 对象的指针, timeRef 是 Time 对象的引用并且 timeObject 已被初始化。

## 6.3 访问结构成员

使用成员访问操作符(圆点操作符和箭头操作符)访问结构(或类)的成员。圆点操作符通过对象变量名或对象的引用访问结构或类。例如,用语句

```
cout << timeObject.hour;
```

打印 timeObject 结构的成员 hour。

用语句

```
cout << timeRef.hour;
```

打印 timeRef 引用的结构的成员 hour。

箭头操作符由负号( - )和大于号( > )组成,中间没有空格,通过对象指针访问结构成员或类成员。假设 timePtr 声明为指向 Time 对象,并将 timeObject 结构的地址赋给 timePtr。要打印指针为 timePtr 的 timeObject 结构的成员 hour,可使用语句

```
timePtr = &timeObject;
cout << timePtr ->hour;
```

表达式 timePtr ->hour 和( \*timePtr).hour 相同,后者复引用指针并且使用圆点操作符访问成员 hour。这里需要加上圆括号,因为圆点操作符(.)的优先级高于指针复引用操作符(\* )。加圆括号和加方括号的箭头操作符和圆点操作符的优先级较高,仅次于第3章介绍的作用域操作符,结合性为从左向右。

**常见编程错误 6.1** 表达式( \*timePtr).hour 指 timePtr 所指 struct 的 hour 成员。省略括号的 \*timePtr 是语法错误,因为“.”的优先级高于“\*”,所以表达式变成\*(timePtr.hour)。这是语法错误,因为指针必须要用箭头引用成员。

## 6.4 用 struct 实现用户自定义类型 Time

图 6.1 中的程序生成用户自定义结构类型 Time,它有 3 个整数成员:hour,minute 和 second。程序定义一个称为 dinnerTime 的 Time 结构,并用圆点操作符初始化结构成员 hour,minute 和 second,3 者的值分别为 18,30 和 0。然后程序按军用格式(或所谓“通用格式”)和标准格式打印时间。注意,打印函数接收常量 Time 结构的引用,从而通过引用把 Time 类型结构传递到打印函数,消除了按值传递打印函数所带来的复制开销,并用 const 关键字防止打印函数修改 Time 结构。第 7 章将介绍常量对象和常量成员函数。

```
1 //Fig. 6.1: fig06_01.cpp
2 //Create a structure, set its members, and print it.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 struct Time { //structure definition
9     int hour; //0-23
10    int minute; //0-59
11    int second; //0-59
12 };
13
14 void printMilitary( const Time & ); //prototype
15 void printStandard( const Time & ); //prototype
16
17 int main()
18 {
19     Time dinnerTime; //variable of new type Time
20
```

```

21 //set members to valid values
22 dinnerTime.hour = 18;
23 dinnerTime.minute = 30;
24 dinnerTime.second = 0;
25
26 cout << "Dinner will be held at ";
27 printMilitary( dinnerTime );
28 cout << " military time,\nwhich is ";
29 printStandard( dinnerTime );
30 cout << " standard time.\n";
31
32 //set members to invalid values
33 dinnerTime.hour = 29;
34 dinnerTime.minute = 73;
35
36 cout << "\nTime with invalid values: ";
37 printMilitary( dinnerTime );
38 cout << endl;
39 return 0;
40 |
41
42 //Print the time in military format
43 void printMilitary( const Time &t )
44 |
45     cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
46         << ( t.minute < 10 ? "0" : "" ) << t.minute;
47 |
48
49 //Print the time in standard format
50 void printStandard( const Time &t )
51 |
52     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
53         12 : t.hour % 12 )
54         << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
55         << ":" << ( t.second < 10 ? "0" : "" ) << t.second
56         << ( t.hour < 12 ? " AM" : " PM" );
57 |

```

输出结果:

Dinner will be held at 18:30 military time,  
which is 6:30:00 PM standard time.

Time with invalid values; 29;73

图 6.1 生成结构,设置结构成员和打印该结构

**性能提示 6.1** 结构通常通过传值调用传递。要避免复制结构的开销,应按引用调用传递结构。

**软件工程知识 6.1** 要避免按传值调用传递的开销而且避免调用者的原始数据被修改,可以将长度较大的参数作为常量引用传递。

用这种方法生成新的数据类型有一定的不足。由于没有指定必须要初始化,所以可能出现未初始化的数据及由此而来的问题。即使数据已经初始化,也可能没有正确初始化。因为程序直接访问数据,所以结构成员可能被赋以无效值(如图 6.1)。在第 33 行和第 34 行,程序很容易将错值赋给 Time 对象 dinnerTime 的成员 hour 和 minute。如果 struct 的实现方法发生变化(例如时间可以表示为从午夜算起的秒数),所有使用 struct 的程序都要改变。因为程序员直接操作数据类型。没有一个“接口”可以确保程序员正确使用数据类型并保持数据的一致性状态。

**软件工程知识 6.2** 编写易于理解且维护的程序很重要。改变是常有的事而不是偶尔为之。程序员应该预料到代码会经常修改。可以看出,类可使程序的更易于修改。

还有其他与 C 语言结构相关的问题。在 C 语言中,结构不能作为一个单位打印,结构成员只能逐个进行打印和格式化。函数应写成以某种适当格式打印结构成员。第 8 章“操作符重载”将演示如何重载 << 操作符,以方便地打印结构类型或类类型的对象。以 C 语言中,结构不能进行整体比较,而只能逐成员地比较。第 8 章还会介绍如何重载操作符与关系操作符,比较 C++ 中的结构类型和类类型的对象。

下一节重新将 Time 结构实现为 C++ 类,并且将演示用类生成所谓抽象数据类型的好处。我们将会看到, C++ 中,类和结构的用法大致相同,两者的差别在于其成员相关的默认访问能力不同。我们将会对此进行简单介绍。

## 6.5 用 class 实现 Time 抽象数据类型

类能让程序员模拟对象的属性(表示为数据成员)和行为或操作(表示为成员函数)。在 C++ 中,用关键字 class 定义包含数据成员和成员函数的类型。

在其他面向对象程序语言中,成员函数有时也称为方法,响应对象接收的信息。信息对应于一个对象发给另一个对象或由函数发给对象的成员函数调用。

类一经定义,其名称就可以用于声明该类的对象。图 6.2 中的程序包含一个 Time 类的简单定义。

```
1 class Time{
2 public:
3     Time();
4     void setTime(int,int,int);
5     void printMilitary();
6     void printStandard();
7 private:
8     int hour;      //0-23
9     int minute;    //0-59
10    int second;    //0-59
11 }
```

图 6.2 Time 类的简单定义

Time 类的定义以关键字 class 开始。类定义体放在左右花括号({})之间,类定义用分号中止。Time 类定义和 Time 结构定义分别包含 3 个整数成员:hour, minute 和 second。

**常见编程错误 6.2** 在类(或结构)定义结尾处不加分号是语法错误。

类定义的其他部分是新内容。`public:`和`private:`标记称为成员访问说明符。在程序能访问`Time`类对象的任何地方,都可以访问成员访问说明符`public`之后、下一个成员访问说明符之前声明的任何数据成员或成员函数。成员访问说明符`private`之后、下一个访问说明符之前所有数据成员或成员函数不同只能由该类的成员函数访问。成员访问说明符之后总有冒号(`:`),并在类定义中以不同顺序多次出现。下文将使用成员访问说明符`public`和`private`(不加冒号)。第9章将介绍第3个成员访问说明符`protected`,以及它与继承在面向对象编程中的作用。

**良好编程习惯 6.1** 为保证定义的清晰性和可读性每个成员访问说明符只在类定义中使用一次。`public`成员会放在前面,以便于寻找。

`public`成员访问说明符之后,类定义包含以下4个成员函数`Time`,`setTime`,`printMilitary`和`printStandard`。这些函数是类的`public`成员函数、`public`服务、`public`行为或类的接口。这些函数供类的客户(程序的用户部分)用于操作类的数据。类的数据成员支持服务的传递,这种服务由类用成员函数提供给类的客户,并且这些服务允许客户与类对象交互。

注意,与类名相同的成员函数叫做该类的构造函数。构造函数是一种特殊的成员函数,它对类对象的数据成员进行初始化。生成类对象时,会自动调用该类的构造函数。一个类往往有几个构造函数,这是通过函数重载完成的。注意,构造函数不指定任何返回类型。

**常见编程错误 6.3** 对构造函数指定返回类型或返回值是语法错误。

成员访问说明符`private`后面有3个整数成员,表示类的成员函数只能由其类成员函数访问,我们还将下一章介绍由类的友元函数进行访问。因此,数据成员只能由函数原型出现在类定义中的4个函数(或类的友元)访问。数据成员通常放在类的`private`部分,成员函数通常放在`public`部分。稍后我们会看到,`private`成员函数或`public`数据可以用,但并不常见,而且也是一个不好的编程习惯。

一旦定义了类,它就可以作为类型用于对象、数组和指针定义

```
Time sunset,                //object of type Time
    arrayOfTimes[5],        //array of Time objects
    *pointerToTime,         //pointer to a Time object
    &dinnerTime = sunset;    //reference to a Time object
```

类名成为新的类型说明符。一个类可以有多个对象,如同`int`类型可以有多个变量一样。程序员可以按需要生成新的类类型。这就是C++被称为扩展语言的原因之一。

图6.3使用了`Time`类。程序实例化`Time`类的一个对象`t`。对象被实例化时,自动调用`Time`构造函数,并将每一个`private`数据成员初始化为0。接着以军用格式和标准格式打印时间,以确认成员已经被正确初始化。然后用`setTime`成员函数设置时间,并再次以上述两种格式打印时间。用`setTime`成员函数试着将数据成员设为无效值,并再次以上述两种格式打印时间。

```
1 //Fig. 6.3: fig06_03.cpp
2 //Time class.
```

```

3  #include <iostream>
4  5  using std::cout;
6  using std::endl;
7  8  //Time abstract data type (ADT) definition
9  class Time {
10 public:
11     Time();                //constructor
12     void setTime( int, int, int ); //set hour, minute, second
13     void printMilitary();    //print military time format
14     void printStandard();   //print standard time format
15 private:
16     int hour;              //0 - 23
17     int minute;           //0 - 59
18     int second;           //0 - 59
19 };
20
21 //Time constructor initializes each data member to zero.
22 //Ensures all Time objects start in a consistent state.
23 Time::Time() { hour = minute = second = 0; }
24
25 //set a new Time value using military time. Perform validity
26 //checks on the data values. set invalid values to zero.
27 void Time::setTime( int h, int m, int s )
28 {
29     hour = ( h >= 0 && h < 24 ) ? h : 0;
30     minute = ( m >= 0 && m < 60 ) ? m : 0;
31     second = ( s >= 0 && s < 60 ) ? s : 0;
32 }
33
34 //Print Time in military format
35 void Time::printMilitary()
36 {
37     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
38           << ( minute < 10 ? "0" : "" ) << minute;
39 }
40
41 //Print Time in standard format
42 void Time::printStandard()
43 {
44     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
45           << ":" << ( minute < 10 ? "0" : "" ) << minute
46           << ":" << ( second < 10 ? "0" : "" ) << second
47           << ( hour < 12 ? " AM" : " PM" );
48 }
49
50 //Driver to test simple class Time
51 int main()
52 {
53     Time t; //instantiate object t of class Time
54
55     cout << "The initial military time is ";

```

```
56 t.printMilitary();
57 cout << "\nThe initial standard time is ";
58 t.printStandard();
59
60 t.setTime( 13, 27, 6 );
61 cout << "\n\nMilitary time after setTime is ";
62 t.printMilitary();
63 cout << "\nStandard time after setTime is ";
64 t.printStandard();
65
66 t.setTime( 99, 99, 99 ); //attempt invalid settings
67 cout << "\n\nAfter attempting invalid settings;"
68     << "\nMilitary time: ";
69 t.printMilitary();
70 cout << "\nStandard time: ";
71 t.printStandard();
72 cout << endl;
73 return 0;
74 }
```

输出结果:

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings;
Military time:00:00
Standard time:12:00:00 AM
```

图 6.3 用 class 实现抽象数据类型 Time

再次提醒大家注意,private 成员访问说明符放在数据成员 hour、minute 和 second 之前。类的 private 数据成员通常不能在类外部访问。第 7 章将介绍,可以用类的友元访问类的 private 成员。可以看出,类的客户并不关心类的实际数据表达。例如,类完全可以用午夜算起的秒数表示时间。客户可以使用相同的 public 成员函数并取得相同的结果而不注意类的变化。从这个意义上讲,类的实现可以视为向客户隐藏起来。这种信息隐藏增强了程序的可修改性,方便了客户对类的理解。

**软件工程知识 6.3** 类客户使用类,可以不知道实现类的内部细节。如果类的实现方法发生改变(例如为了提高性能),只要类的接口保持不变,类的客户源代码就不必变(尽管客户可能需要重新编译),这就使修改系统变得更加容易。

这个程序中,Time 构造函数将数据成员初始化为 0(即军用时间上午 12 时)。这保证了生成对象时,对象会保持一致。无效值不能存放在 Time 对象的数据成员里,因为生成 Time 对象时,会自动调用构造函数,而且客户对数据成员的后期修改都是由函数 setTime 完成的。

**软件工程知识 6.4** 成员函数通常比非面向对象程序中的函数短,因为数据成员中保存的数据已经由构造函数和保存新数据的成员函数验证。由于数据已经是对象,成员函数调用通常不带参数或参数个数至少比非面向对象语言中的典型函数调用的参数少。所以,调用更简短、函数定义更简化,函数原型也更简化。

注意,类的数据成员不能在类体中声明时初始化。而是用类的构造函数初始化,或者用“设置”(set)函数为其赋值。

**常见编程错误 6.4** 试图将类定义中显式初始化类的数据成员是语法错误。

与类同名,但前面添加了代字符(~)的函数被称为该类的析构函数(本例没有显式包含析构函数,所以系统会插入一个析构函数)。析构函数在系统收回对象的内存之前清理每个类对象。析构函数不能带参数,所以不能重载。本章稍后和第7章将详细介绍构造函数和析构函数。

注意,类向外部提供的函数之前要加上 public 成员访问说明符。public 函数实现类提供给客户的行为或服务,通常称为类的接口或 public 接口。

**软件工程知识 6.5** 客户可以访问类的接口,但不能访问类的实现方法。

类定义包含类的数据成员和类的成员函数的声明。成员函数声明就是前文介绍的函数原型。成员函数可以在类的内部定义,但在类的外部定义是良好的编程习惯。

**软件工程知识 6.6** 通过函数原型在类定义中声明成员函数,在类定义之外定义这些函数,可以区分类的接口与实现方法。这样可以实现良好的软件工程。类的客户看不到该类成员函数的实现方法,即使实现方法发生改变,也不需要重新编译。

注意,图 6.3 中的类定义中,每个成员函数定义都使用了二元作用域分辨符(::)。一旦定义了类并声明了成员函数,就必须定义成员函数。类的每个成员函数可以直接在类定义体中定义(而不是包括类的函数原型),或者可以在类定义体之后定义成员函数。在类定义之后定义成员函数时,成员名之前要加上类名和二元作用域分辨符(::)。因为不同类可以有相同的成员名,所以要用二元作用域分辨符将成员名与类名联系起来,惟一标识某个类的成员函数。

**常见编程错误 6.5** 在类外部定义类的成员函数时,忽略函数名中的类名和作用域分辨符是错误的。

尽管类定义中声明的成员函数可以在类定义外部定义,但成员函数仍在类的作用域中,即只有类的其他成员知道它的名称,否则就须通过类对象、引用类对象或类对象指针进行引用。稍后将详细介绍类的作用域。

如果在类定义中定义成员函数,该成员函数就会自动成为内联函数。在类定义外部定义的成员函数可通过关键字 inline 显式指定为内联函数。记住,编译器有权不在程序块中保留内联函数。

**性能提示 6.2** 在类定义中定义小的成员函数将自动使该成员函数成为内联函数(如果编译器如此选择的话)。这样可以提高性能,但是不能提高软件工程质量,因为类客户能看到



函数的实现方法,并且如果内联函数定义发生了变化,就必须重新编译代码。

**软件工程知识 6.7** 只有最简单的成员函数和最稳定的成员函数(即实现方法不会轻易改变)可在类的头文件中定义。

有趣的是,成员函数 `printMilitary` 和 `printStandard` 没有参数。这是因为成员函数隐式知道要打印调用的特定 `Time` 对象的数据成员。这就使成员函数调用比过程式中的传统函数调用更简练。

**测试和调试提示 6.1** 实际上,成员函数调用通常不带参数或其参数个数比非面向对象语言中的传统函数调用参数要少,其目的是减少传递错误参数、错误参数类型或错误参数个数的可能性。

**软件工程知识 6.8** 使用面向对象编程方法通常可以减少传递的参数个数,以简化函数调用。面向对象编程好处是由于对象中封装了数据成员和成员函数,成员函数有权访问数据成员。

类可简化编程,因为客户(或者类对象的用户)只须关心对象的封装或嵌入操作。这种操作通常是面向客户设计的,而不是面向实现的方法。客户不需要关心类的实现方法(尽管客户需要正确有效的实现方法)。接口发生改变,但不像实现方法那样频繁。实现方法改变时,与其有关的代码也需改变。隐藏实现方法后,可消除程序中与类实现方法细节有关的代码。

**软件工程知识 6.9** 本书的重点是“复用、复用、再复用”。我们将看重介绍几个提高复用性的技术。重点将放在“建立宝贵的类”和建立宝贵的“软件资产”。

类通常不需要从头创建,可以从其他供新类使用的属性和行为的类派生而来。类可以包括其他类对象,将其作为成员。这种软件复用可以大大提高程序员的工作效率。从现成的类中派生将新类称为继承,第9章详细介绍继承。包含其他类对象作为成员称为复合,详情参见第7章。

才接触面向对象编程的人常常担心对象会很大,因为其中要包含数据和函数。从逻辑上讲,的确如此,程序员可以把对象视为包含数据和函数,然而事实却相反。

**性能提示 6.3** 实际上,对象只包含数据,所以比其中包含函数的对象要小得多。对类名或该类的对象采用 `sizeof` 操作符时,只能得到类的数据的长度。编译器生成独立于所有类的对象的成员函数的副本(只有一份)。类的所有对象共享这个成员函数的惟一副本。当然,每个对象都需要自己的类数据副本,因为对象中的数据是不同的。函数代码是不可修改的(也称为可重入码或纯过程),因而可供类的所有对象共享。

## 6.6 类作用域和访问类成员

类的数据成员(在类定义中声明的变量)和成员函数(在类定义中声明的函数)属于该类的作用域。非成员函数在文件作用域中定义。

在类作用域中,类成员由该类的所有成员函数直接访问,也可以用名称引用。在类的作

用域之外,类成员通过对象的句柄(可以是对象名、对象引用或对象指针)引用。第7章将介绍,每次引用对象中的数据成员或成员函数时,编译器会插入一个隐式句柄。

类的成员函数可以重载,但是只能由该类的其他成员函数重载。要重载一个成员函数,只须在类定义中提供该重载函数每个版本的原型,并对每个重载函数的版本提供不同的函数定义。

在成员函数中定义的变量有函数作用域,只有该函数知道函数作用域。如果成员函数定义了与类作用域内变量同名的另一个变量,那么在函数作用域中,函数作用域内的变量将隐藏类作用域内的变量。要访问这种隐藏变量只须在其前面添加类名和作用域分辨符(::)。隐藏的全局变量可以用一元操作符访问(参见第3章)。

用于访问类成员操作符与用于访问结构成员的操作符相同。圆点成员选择操作符(.)与对象名或对象引用结合使用,即可访问对象成员。箭头成员选择操作符(->)与对象指针结合使用,即可访问对象成员。

图6.4中的程序用简单的Count类和public数据成员x(int类型)以及public成员函数print演示用成员选择操作符访问类成员。程序生成(定义)3个Count类型的变量——counter、counterRef(Count对象的引用)和counterPtr(Count对象的指针)。变量counterRef定义为引用counter、变量counterPtr定义为指向counter。注意,这里将数据成员x设为public,目的仅是演示如何利用句柄(如名称、引用或指针)访问public成员。如前所述,数据通常指定为private。第9章将介绍某些情况下可以将数据指定为protected。

```

1 //Fig. 6.4: fig06_04.cpp
2 //Demonstrating the class member access operators . and ->
3 //
4 //CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
5 #include <iostream>
6
7 using std::cout;
8 using std::endl;
9
10 //Simple class Count
11 class Count {
12 public:
13     int x;
14     void print() { cout << x << endl; }
15 };
16
17 int main()
18 {
19     Count counter,           //create counter object
20         *counterPtr = &counter, //pointer to counter
21         &counterRef = counter; //reference to counter
22
23     cout << "Assign 7 to x and print using the object's name: ";
24     counter.x = 7;           //assign 7 to data member x
25     counter.print();         //call member function print
26

```

```

27  cout << "Assign 8 to x and print using a reference: ";
28  counterRef.x = 8;           //assign 8 to data member x
29  counterRef.print();        //call member function print
30
31  cout << "Assign 10 to x and print using a pointer: ";
32  counterPtr ->x = 10; //assign 10 to data member x
33  counterPtr ->print(); //call member function print
34  return 0;
35 |

```

输出结果:

```

Assign 7 to x and print using the object's name:7
Assign 8 to x and print using a reference:8
Assign 10 to x and print using a pointer:10

```

图 6.4 通过各种对象句柄(对象名称、引用和对象指针)访问对象数据成员和成员函数

## 6.7 接口同实现方法的分离

良好软件工程的基本原则之一是将接口与实现方法分离,这样可以使程序更易于修改。类实现方法的改变不会对类的客户造成影响,只要提供给客户的类的原接口不变(类的功能可能扩展至原接口以外)。

**软件工程知识 6.10** 将类声明放在使用该类的任何客户的头文件中,形成类的 public 接口(并向客户提供调用类成员函数的函数原型)。将类成员函数的定义放在源文件中,形成类的实现方法。

**软件工程知识 6.11** 类客户如果要使用类,不需要访问类的源代码。但是,客户需要连接类的对象代码。这样便由独立软件供应商(ISV)提供类库进行销售或发放许可证。独立软件供应商在自己的产品中只提供头文件和目标模块,不提供专属信息(例如源代码)。C++ 用户群可以享用更多独立软件供应商生产的类库。

事实上,任何事情都不是十全十美的。头文件中的确包含部分实现细节,并隐藏实现的其他部分。例如,内联成员函数需要放在头文件中,如此来,编译器编译客户代码时,该客户代码可以包含内联函数定义。private 成员列在头文件的类定义中,所以尽管客户不能访问 private 成员,但能看到它们。第 7 章将介绍如何用所谓的代理类在类的客户中隐藏类的 private 数据。

**软件工程知识 6.12** 类接口的关键信息应放在头文件中。只用于类内部使用且类客户不需要的信息应放在不发布的源文件中。这是最低权限原则的另一个例子。

图 6.5 中的程序将图 6.3 中的程序分解成多个文件。建立一个 C++ 程序时,各个类定义通常包含在头文件中,类的成员函数定义包含在基本名字相同的源代码文件中。使用了类的每个文件中都包含头文件,这是通过 #include 来实现的,源代码文件编译并连接包含主程序的文件。查看编译器文档,以确定如何编译和连接其中包含多个源文件的程序。

```

1 //Fig. 6.5: time1.h
2 //Declaration of the Time class.
3 //Member functions are defined in time1.cpp
4
5 //prevent multiple inclusions of header file
6 #ifndef TIME1_H
7 #define TIME1_H
8
9 //Time abstract data type definition
10 class Time {
11 public:
12     Time(); //constructor
13     void setTime( int, int, int ); //set hour, minute, second
14     void printMilitary(); //print military time format
15     void printStandard(); //print standard time format
16 private:
17     int hour; //0 - 23
18     int minute; //0 - 59
19     int second; //0 - 59
20 };
21
22 #endif

```

图 6.5 分离 Time 类的接口与实现方法——time1.h

```

23 //Fig. 6.5: time1.cpp
24 //Member function definitions for Time class.
25 #include <iostream>
26
27 using std::cout;
28
29 #include "time1.h"
30
31 //Time constructor initializes each data member to zero.
32 //Ensures all Time objects start in a consistent state.
33 Time::Time() { hour = minute = second = 0; }
34
35 //set a new Time value using military time. Perform validity
36 //checks on the data values. set invalid values to zero.
37 void Time::setTime( int h, int m, int s )
38 {
39     hour = ( h >= 0 && h < 24 ) ? h : 0;
40     minute = ( m >= 0 && m < 60 ) ? m : 0;
41     second = ( s >= 0 && s < 60 ) ? s : 0;
42 }
43
44 //Print Time in military format
45 void Time::printMilitary()
46 {
47     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
48         << ( minute < 10 ? "0" : "" ) << minute;
49 }

```

```

50
51 //Print time in standard format
52 void Time::printStandard()
53 {
54     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
55         << ":" << ( minute < 10 ? "0" : "" ) << minute
56         << ":" << ( second < 10 ? "0" : "" ) << second
57         << ( hour < 12 ? " AM" : " PM" );
58 }

```

图 6.5 分离 Time 类的接口与实现方法——time1.cpp

```

59 //Fig. 6.5: fig06_05.cpp
60 //Driver for Time1 class
61 //NOTE: Compile with time1.cpp
62 #include <iostream>
63
64 using std::cout;
65 using std::endl;
66
67 #include "time1.h"
68
69 //Driver to test simple class Time
70 int main()
71 {
72     Time t; //instantiate object t of class time
73
74     cout << "The initial military time is ";
75     t.printMilitary();
76     cout << "\nThe initial standard time is ";
77     t.printStandard();
78
79     t.setTime( 13, 27, 6 );
80     cout << "\n\nMilitary time after setTime is ";
81     t.printMilitary();
82     cout << "\nStandard time after setTime is ";
83     t.printStandard();
84
85     t.setTime( 99, 99, 99 ); //attempt invalid settings
86     cout << "\n\nAfter attempting invalid settings:\n"
87         << "Military time: ";
88     t.printMilitary();
89     cout << "\nStandard time: ";
90     t.printStandard();
91     cout << endl;
92     return 0;
93 }

```

输出结果:

```

The initial military time is 00:00
The initial standard time is 12:00:00 AM

```

```

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time:00:00
Standard time:12:00:00 AM

```

图 6.5 分离 Time 类的接口与实现方法——fig06\_05.cpp

图 6.5 中的程序包含头文件 `time1.h`, 类 `Time` 在该头文件中声明, 类 `Time` 的成员函数在文件 `time1.cpp` 中定义, 函数 `main` 在文件 `fig06_05.cpp` 中定义。该程序的输出和图 6.3 中程序的输出相同。

注意, 类声明封装在预处理程序

```

//prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME_H
...
#endif

```

中。建立比较大的程序时, 其他定义和声明也被放在头文件中。如果已经定义了 `TIME1_H` 名, 上述预处理程序指令不再包含 `#ifndef` (如果没有定义的话) 和 `#endif` 之间的代码。如果文件中本来没有包含头文件, 则说明 `TIME1_H` 的名称已经由 `#define` 指令定义; 如果文件中已经包含头文件, 则说明 `TIME1_H` 已经定义, 不再包含头文件。多次包含头文件语句的情况通常发生在大程序中, 许多头文件本身已包含其他头文件。注意, 预处理程序指令中符号化常量名使用的规则是把头文件名中的圆点符号换成下划线。

**测试和调试提示 6.2** 利用 `#ifndef`, `#define` 和 `#endif` 预处理程序指令防止一个程序中多次包含头文件。

**良好编程习惯 6.2** 头文件的 `#ifndef` 和 `#define` 预处理程序指令中使用头文件名, 并用下划线代替圆点。

## 6.8 控制对成员的访问

成员访问说明符 `public` 和 `private` (以及第 9 章介绍的 `protected`) 用于控制对类数据成员和成员函数的访问。类的默认访问模式是 `private`, 所以类首部和第一个成员访问说明符之间的所有成员都是 `private` 类型。每个成员访问说明符之后, 采用成员访问说明符调用的模式, 直到遇到下一个成员访问说明符或遇到类定义的中止右花括号 (} )。成员访问说明符 `public`、`private` 和 `protected` 可重复使用, 但是较为少见, 因为容易造成混乱。

只有类的成员函数 (与第 7 章介绍的友元) 可访问类的 `private` 成员。类的 `public` 成员则可以由程序中的任何函数访问。

`public` 成员的主要用途是向类的客户提供类的服务 (行为)。这组服务形成类的 `public` 接口。类的客户无须关心类的实现细节。类的客户不能访问类的 `private` 成员和 `public` 成员函数的定义。这些组件形成类的实现方法。

**软件工程知识 6.13** C++ 提倡程序独立于实现方法。独立代码使用的类实现方法发生改变时,无需去改变代码。类接口的任何部分一旦发生改变,独立于实现方法的代码就必须重新编译。

**常见编程错误 6.6** 不是特定类的成员函数(或类的友元)试图访问类的 private 成员是语法错误。

图 6.6 演示了 private 类成员只能用 public 类接口通过 public 成员函数访问。编译程序时,编译器生成两条错误消息,表示无法访问每条语句中指定的 private 成员。图 6.6 包含 timel.h,并且随图 6.5 中的 timel.cpp 一起编译。

```

1 //Fig. 6.6; fig06_06.cpp
2 //Demonstrate errors resulting from attempts
3 //to access private class members.
4 #include <iostream>
5
6 using std::cout;
7
8 #include "timel.h"
9
10 int main()
11 {
12     Time t;
13
14     //Error: 'Time::hour' is not accessible
15     t.hour = 7;
16
17     //Error: 'Time::minute' is not accessible
18     cout << "minute = " << t.minute;
19
20     return 0;
21 }
```

Borland C++ 命令行编译器输出的错误消息:

```

Timel.cpp:
Fig06_06.cpp:
Error E2247 Fig06_06.cpp 15:
    'Time:hour' is not accessible in function main()
Error E2247 Fig06_06.cpp 18:
    'Time::minute' is not accessible in function main()
```

\*\*\*2 errors in Compile\*\*\*

Microsoft Visual C++ 编译器输出的错误消息:

```

Compiling...
Fig06_06.cpp
D:\Fig06_06.cpp(15):error C2248:'hour';cannot access private
member declared in class 'Time'
D:\Fig6_06\timel.h(18):see declaration of 'hour'
```

```
D:\Fig06_06.cpp(18):error C2248:'minute':cannot access private
member declared in class 'Time'
D:\time1.h(19):see declaration of 'minute'
Error excuting cl.exe.
```

```
test.exe -2 error(s),0 warning(s)
```

图 6.6 错误访问类的 private 成员

**良好编程习惯 6.3** 如果可以选择在类定义中先列出 private 成员,尽管程序默认访问模式为 private,但最好显式使用 private 成员访问说明符,这样可使程序更清晰。我们提倡先列出类的 public 成员以强调类的接口。

**良好编程习惯 6.4** 尽管 public 和 private 成员访问说明符可以重复和混用,但最好先将所有类的 public 成员列成一组,然后再将所有类的 private 成员列成一组。这样可使用户更关注类的 public 接口,而不是类的实现方法。

**软件工程知识 6.14** 尽量使所有类的数据成员保持 private。提供 public 函数,设置 private 数据成员的值,并获得 private 数据成员的值。这种结构能够向类的客户隐藏实现方法,在减少错误的同时,还增强了程序的可修改性。

类的客户可以是另一个类的成员函数,或者是全局函数(例如文件中类 C 语言的“松散”函数或“自由”函数,它们不是任何类的成员函数)。

类成员的默认访问方式是 private。类成员的访问方式可以显式设置为 public,protected (参见第 9 章)或 private。struct 成员的默认访问方式是 public。struct 成员的访问方式也可以显式设置为 public,protected 或者 private。

**软件工程知识 6.15** 类设计人员用 private,protected 和 public 成员实施隐藏信息的概念和最低权限原则。

类数据为 private 并不表示客户代码不能修改这类数据。可以通过该类的成员函数或友元函数修改这类数据。稍后会提到,设计这些函数时应确保数据的完整性。

应该用访问函数(也称为访问方法)的成员函数谨慎控制类的 private 数据的访问方式。例如,要让客户读取 private 数据的值,类可以提供一个 get(获得)函数。要使客户修改 private 数据,类可以提供 set(设置)函数。这样的修改似乎违背了 private 数据的概念,但是 set 函数可以提供数据确认功能(如范围检查)以保证正确地设置数值。set 函数可以在接口中使用的数据形式和实现方法中使用的数据形式之间进行转换。get 函数不需要以“原始”形式显示数据,该函数可以编辑数据,并且可以限制客户看到的数据。

**软件工程知识 6.16** 类设计人员不需要提供每个 private 数据成员的 set 和 get 函数,只在需要的时候才提供数据成员的 set 和 get 函数。如果服务对于客户代码是有用的,则应该在类的 private 接口中提供服务。

**测试和调试提示 6.3** 将类的数据成员指定为 private、类的成员函数指定为 public 有助于调试,因为数据操作在类成员函数中或类的友元函数中局部进行。



## 6.9 访问函数和工具函数

并非所有成员函数都要指定为 `public`, 才能充当类的接口的一部分。一些成员函数仍然保留 `private`, 充当类中其他函数的工具函数。

**软件工程知识 6.17** 成员函数分为不同类别: 读取和返回 `private` 数据成员值的函数、设置 `private` 数据成员值的函数、实现类的服务的函数和进行各种类操作的函数, 例如初始化类对象、赋值类对象、将类与内部类型和其他类进行相互转换以及处理类对象内存。

访问函数可以读取或显示数据。访问函数另一个常见用法是测试条件的真假, 这类函数通常称为判定函数。任何容器类都有的 `isEmpty` 函数(如链表、堆栈和队列)就是判定函数。程序先测试 `isEmpty`, 再从容器对象中读取另一个项目。`isFull` 判定函数可以测试容器类对象以确定容器类对象是否还有多余的空间。`Time` 类的一组有用的判断函数可以包括 `isAM` 和 `isPM`。

图 6.7 演示了工具函数(或称为帮助函数)的概念。工具函数不属于类的接口, 而是 `private` 成员函数, 支持类的 `public` 成员函数的操作。类的客户不能使用工具函数。

```

1 //Fig. 6.7: salesp.h
2 //SalesPerson class definition
3 //Member functions defined in salesp.cpp
4 #ifndef SALESP_H
5 #define SALESP_H
6
7 class SalesPerson {
8 public:
9     SalesPerson();           //constructor
10    void getSalesFromUser();  //get sales figures from keyboard
11    void setSales( int, double ); //User supplies one month's
12                                //sales figures.
13    void printAnnualSales();
14
15 private:
16    double totalAnnualSales(); //utility function
17    double sales[ 12 ];        //12 monthly sales figures
18 };
19
20 #endif

```

图 6.7 工具函数用法示例——`salesp.h`

```

21 //Fig. 6.7: salesp.cpp
22 //Member functions for class SalesPerson
23 #include <iostream>
24
25 using std::cout;
26 using std::cin;
27 using std::endl;

```

```
28
29 #include <iomanip>
30
31 using std::setprecision;
32 using std::setiosflags;
33 using std::ios;
34
35 #include "salesp.h"
36
37 //Constructor function initializes array
38 SalesPerson::SalesPerson()
39 {
40     for ( int i = 0; i < 12; i ++ )
41         sales[ i ] = 0.0;
42 }
43
44 //Function to get 12 sales figures from the user
45 //at the keyboard
46 void SalesPerson::getSalesFromUser()
47 {
48     double salesFigure;
49
50     for ( int i = 1; i <= 12; i ++ > |
51 cout << "Enter sales amount for month " << i << ": ";
52
53     cin >> salesFigure;
54     setSales( i, salesFigure );
55 }
56 |
57
58 //Function to set one of the 12 monthly sales figures.
59 //Note that the month value must be from 0 to 11.
60 void SalesPerson::setSales( int month, double amount )
61 {
62     if ( month >= 1 && month <= 12 && amount > 0 )
63         sales[ month - 1 ] = amount; //adjust for subscripts 0 -11
64     else
65         cout << "Invalid month or sales figure" << endl;
66 }
67
68 //Print the total annual sales
69 void SalesPerson::printAnnualSales()
70 {
71     cout << setprecision( 2 )
72         << setiosflags( ios::fixed | ios::showpoint )
73         << "\nThe total annual sales are: $"
74         << totalAnnualSales() << endl;
75 }
76
77 //Private utility function to total annual sales
78 double SalesPerson::totalAnnualSales()
```

```

79 |
80     double total = 0.0;
81
82     for ( int i = 0; i < 12; i ++ >
83         total += sales[ i ];
84
85     return total;
86 |

```

图 6.7 工具函数用法示例——salesp.cpp

SalesPerson 类中代表 12 个月销售数据的数组用构造函数初始化为 0,并且用 setSales 设置为用户提供的值。public 成员函数 printAnnualSales 打印最近 12 个月的总销售额。工具函数 totalAnnualSales 为 printAnnualSales 计算 12 个月的总销售额。成员函数 printAnnualSales 将销售额编辑成美元金额的格式。

```

87 //Fig. 6.7; fig06_07.cpp
88 //Demonstrating a utility function
89 //Compile with salesp.cpp
90 #include "salesp.h"
91
92 int main()
93 {
94     SalesPerson s;           //create SalesPerson object s
95
96     s.getSalesFromUser();     //note simple sequential code
97     s.printAnnualSales();     //no control structures in main
98     return 0;
99 |

```

输出结果:

```

Enter sales amount for month 1:5314.76
Enter sales amount for month 2:4292.38
Enter sales amount for month 3:4589.83
Enter sales amount for month 4:5534.03
Enter sales amount for month 5:4376.34
Enter sales amount for month 6:5698.45
Enter sales amount for month 7:4439.22
Enter sales amount for month 8:5893.57
Enter sales amount for month 9:4909.67
Enter sales amount for month 10:5123.45
Enter sales amount for month 11:4024.97
Enter sales amount for month 12:5923.92

```

```
The total annual sales are: $ 60120.59
```

图 6.7 工具函数用法示例——fig06\_07.cpp

注意,main 中只包含一个简单的成员函数调用,不包含任何控制结构。

**软件工程知识 6.18** 面向对象编程的一个现象是,一旦定义了类,生成和操作该类的对象通常只要一个简单的成员函数调用,不需要或只需要少量控制结构。类成员函数的实现方

法则不然,通常包含控制结构。

## 6.10 初始化类对象:构造函数

生成类对象时,该类的成员可以由类的构造函数初始化。构造函数是与类同名的类成员函数。程序员提供的构造函数,每次生成类对象(实例化)的时候自动调用这个构造函数。构造函数可以重载,提供初始化类成员的不同访问。数据成员必须在类的构造函数中初始化或者在生成对象之后设置其数值。但是,一个良好的编程习惯和软件工程知识可以保证对象在客户代码调用其成员函数之前完成初始化。一般说来,不能依靠客户代码来保证对象的正确初始化。

**常见编程错误 6.7** 类的数据成员不能在类定义中初始化。

**常见编程错误 6.8** 试图声明构造函数的返回类型和/或试图从构造函数中返回值是语法错误。

**良好编程习惯 6.5** 适当的时候(几乎一直如此)提供构造函数可保证各个对象正确初始化为有意义的值。指针数据成员尤其如此,应初始化为某个合法指针值或初始化为0。

**测试和调试提示 6.4** 每个修改对象 private 数据成员的成员函数(及其友元函数)都应保证该数据的一致性。

声明类对象时,可以在对象名之后和分号之前的括号中提供初始化值。这些初始化值作为参数传递到类的构造函数。稍后将介绍几个构造函数调用示例。注意,虽然程序不显式调用构造函数,但程序员仍可提供数据,将其作为参数传递给构造函数。

## 6.11 在构造函数中使用默认参数

timel.cpp(参见图6.5)中的构造函数将 hour, minute 和 secont 初始化为0(即军用时间午夜12时)。构造函数可以包含默认参数。图6.8中的程序重新定义 Time 构造函数,该函数中各变量的默认参数为0。提供后构造函数默认参数,即使构造函数调用中不提供任何数值,仍然可保证对象初始化为一致状态。程序员提供默认所有参数(或显式不需要参数)的构造函数也是默认构造函数。例如,不用参数调用的构造函数。每个类只有一个默认构造函数。

```
1 //Fig. 6.8: time2.h
2 //Declaration of the Time class.
3 //Member functions are defined in time2.cpp
4
5 //preprocessor directives that
6 //prevent multiple inclusions of header file
7 #ifndef TIME2_H
8 #define TIME2_H
9
10 //Time abstract data type definition
```

```

11 class Time {
12 public:
13     Time( int = 0, int = 0, int = 0 ); //default constructor
14     void setTime( int, int, int ); //set hour, minute, second
15     void printMilitary(); //print military time format
16     void printStandard(); //print standard time format
17 private:
18     int hour; //0 - 23
19     int minute; //0 - 59
20     int second; //0 - 59
21 };
22
23 #endif

```

图 6.8 使用带默认参数的构造函数——time2.h

```

24 //Fig. 6.8: time2.cpp
25 //Member function definitions for Time class.
26 #include <iostream>
27
28 using std::cout;
29
30 #include "time2.h"
31
32 //Time constructor initializes each data member to zero.
33 //Ensures all Time objects start in a consistent state.
34 Time::Time( int hr, int min, int sec )
35     { setTime( hr, min, sec ); }
36
37 //set a new Time value using military time. Perform validity
38 //checks on the data values. set invalid values to zero.
39 void Time::setTime( int h, int m, int s )
40 {
41     hour = ( h >= 0 && h < 24 ) ? h : 0;
42     minute = ( m >= 0 && m < 60 ) ? m : 0;
43     second = ( s >= 0 && s < 60 ) ? s : 0;
44 }
45

```

图 6.8 使用带默认参数的构造函数——time2.cpp

```

1 //Print Time in military format
2 void Time::printMilitary()
3 {
4     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
5         << ( minute < 10 ? "0" : "" ) << minute;
6 }
7
8 //Print Time in standard format
9 void Time::printStandard()
10 {
11     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
12         << ":" << ( minute < 10 ? "0" : "" ) << minute

```

```

13         << ":" << ( second < 10 ? "0" : "" ) << second
14         << ( hour < 12 ? " AM" : " PM" );
15 |

```

图 6.8 使用带默认参数的构造函数——time2. cpp

```

16 //Fig. 6.8: fig06_08.cpp
17 //Demonstrating a default constructor
18 //function for class Time.
19 #include <iostream>
20
21 using std::cout;
22 using std::endl;
23
24 #include "time2.h"
25
26 int main()
27 |
28     Time t1,           //all arguments defaulted
29         t2(2),         //minute and second defaulted
30         t3(21, 34),    //second defaulted
31         t4(12, 25, 42), //all values specified
32         t5(27, 74, 99); //all bad values specified
33
34     cout << "Constructed with:\n"
35         << "all arguments defaulted;\n ";
36     t1.printMilitary();
37     cout << "\n ";
38     t1.printStandard();
39
40     cout << "\nhour specified; minute and second defaulted:"
41         << "\n ";
42     t2.printMilitary();
43     cout << "\n ";
44     t2.printStandard();
45
46     cout << "\nhour and minute specified; second defaulted:"
47         << "\n ";
48     t3.printMilitary();
49     cout << "\n ";
50     t3.printStandard();
51
52     cout << "\nhour, minute, and second specified:"
53         << "\n ";
54     t4.printMilitary();
55     cout << "\n ";
56     t4.printStandard();
57
58     cout << "\null invalid values specified:"
59         << "\n ";
60     t5.printMilitary();
61     cout << "\n ";

```

```

62     t5.printStandard();
63     cout << endl;
64
65     return 0;
66 }

```

输出结果:

```

Constructed with:
all arguments defaulted:
    00:00
    12:00:00 AM
hour specified;minute and second defaulted:
    02:00
    2:00:00 AM
hour and minute specified;second defaulted:
    21:34
    9:34:00 PM
hour,minute,and second specified:
    12:25
    12:25:42 PM
all invalid values specified:
    00:00
    12:00:00 AM

```

图 6.8 使用带默认参数的构造函数——fig06\_08.cpp

这个程序中,构造函数调用成员函数 setTime,将数值传递到构造函数(或默认函数),保证 hour 值在 0~23 之间取值、minute 和 second 值在 0~59 之间取值。如果数值超出这里的范围, setTime 就会将其设置为 0(这是保证数据成员一致性的例子之一)。

注意,编写 Time 构造函数时,可以在其中包含与成员函数 setTime 相同的语句。这样可稍微提升程序的性能,因为无须另行调用 setTime 函数。但是,用相同的代码编写 Time 构造函数和成员函数 setTime 加大维护程序的难度。如果成员函数 setTime 的实现方法改变, Time 构造函数的实现方法也应相应改变。使用 Time 构造函数直接调用 setTime 时, setTime 成员函数的实现方法只需改变一次,这样就可以在改变实现方法时减少程序错误。另外,显式声明内联构造函数或在类定义中定义构造函数也可以提高 Time 构造函数的性能(后者隐式包含内联函数定义)。

**软件工程知识 6.19** 如果类的成员函数已经提供类构造函数(或其他成员函数)所需的全部或部分功能,就可以从构造函数(或其他成员函数)中调用成员函数。这不仅简化了代码的维护,还减少了修改代码实现方法时出错的可能性。因此,避免重复代码便成了普遍原则。

**良好编程习惯 6.6** 只在头文件中类定义内部的函数原型中声明默认函数参数值。

**常见编程错误 6.9** 在头文件和或员函数定义中指定同一成员函数的默认初始化值。

注意,对方法默认参数的任何修改都要求重新编译客户代码。如果默认参数值会发生改变,就要换用重载函数。因此,成员函数的实现方法一旦改变,就需要重新编译客户代码。

图 6.8 中的程序初始化 5 个 Time 对象:一个在构造函数调用中将 3 个参数指定为默认值;一个指定一个参数;一个指定两个参数;一个指定 3 个参数;一个指定 3 个无效参数。实例化和初始化之后显示每个对象数据成员的内容。

如果不定义类的构造函数,编译器会生成一个默认构造函数。这种构造函数不执行任何初始化操作,所以生成对象时,无法保证一致状态。

**软件工程知识 6.20** 如果定义了构造函数,并且没有显式的默认构造函数,类就不一定有默认构造函数。

## 6.12 使用析构函数

析构函数是一种特殊的类成员函数。类的析构函数名由类名称和代字符(~)组成。这种命名规则很直观,因为本章稍后将说明代字符是按位取字符,并且从这个意义上看,析构函数是构造函数的反函数。

删除对象时就会调用类的析构函数,即程序执行离开初始化类对象的作用域时对类对象执行的自动操作。事实上,析构函数本身并不是要删除对象,而是在系统回收对象内存之前执行的结束清理工作,以便内存可重复用于保存新对象。

构造函数不接收参数也不返回数值。类只能有一个析构函数,而且析构函数不允许重载。

**常见编程错误 6.10** 向析构函数传递参数、指定析构函数的返回值类型(即使指定了 void)以及从析构函数返回数值或重载析构函数都是语法错误。

注意,此前介绍的类没有提供析构函数。随后将介绍几个使用析构函数的例子。第 8 章将介绍析构函数适用于动态分配内存的对象类(例如数组和字符串)。第 7 章将介绍如何动态分配和释放内存。

**软件工程知识 6.21** 本书稍后介绍的构造函数和析构函数在 C++ 和面向对象编程中非常重要,这里的简短介绍尚不能充分说明。

## 6.13 何时调用构造函数和析构函数

构造函数和析构函数是自动调用的。这些函数的调用顺序由执行过程进入和离开对象实例化的作用域的顺序决定。通常,析构函数的调用顺序与构造函数相反。但是,如图 6.9 中的程序所示,对象存储类可以改变调用析构函数的顺序。

全局范围内定义的对象构造函数在文件中的其他函数(包括 main)开始执行前调用(尽管不能确定全局对象的构造函数的执行顺序)。当 main 中止或调用 exit 函数(详见第 18 章),调用相应的析构函数。如果通过调用 abort 函数(详见第 18 章)中止程序,则不调用全局对象析构函数。

程序执行到定义对象时,调用自动局部对象的构造函数。对象离开作用域(即离开定义对象的块)时,调用相应的析构函数。每次对象进入和离开作用域时,调用自动对象的构造



函数和析构函数。如果想调用 `exit` 和 `abort` 函数来中止程序,可以不调用自动对象的析构函数。

程序执行到首次定义对象时,只调用静态局部对象的构造函数一次。`main` 中止或调用 `exit` 函数时,调用相应的析构函数。如果想通过调用函数 `abort` 中止程序,可以不调用静态对象的析构函数。

图 6.9 中的程序演示了 `CreateAndDestroy` 类型的对象在几种作用域中调用构造函数和析构函数顺序。程序在全局作用域中定义 `first`,在程序开始执行时调用它的构造函数,在删除其他对象之后程序中中止时调用它的析构函数。

函数 `main` 声明 3 个对象。对象 `second` 和 `fourth` 是局部自动对象,对象 `third` 是静态局部对象。当程序执行到声明对象时,调用这些对象的构造函数。当程序执行到 `main` 的结尾时,依次调用对象 `fourth` 和 `second` 的析构对象。因为对象 `third` 是 `static` 局部对象,所以直到程序中中止时才退出。对象 `third` 的析构函数在对象 `first` 的析构对象之前,删除所有其他对象之后调用。

```

1 //Fig. 6.9; create.h
2 //Definition of class CreateAndDestroy.
3 //Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8 public:
9     CreateAndDestroy( int ); //constructor
10    ~CreateAndDestroy(); //destructor
11 private:
12    int data;
13 };
14
15 #endif

```

图 6.9 调用构造函数和析构函数的顺序——`create.h`

```

16 //Fig. 6.9; create.cpp
17 //Member function definitions for class CreateAndDestroy
18 #include <iostream>
19
20 using std::cout;
21 using std::endl;
22
23 #include "create.h"
24
25 CreateAndDestroy::CreateAndDestroy( int value )
26 {
27     data = value;
28     cout << "Object " << data << " constructor";
29 }
30
31 CreateAndDestroy::~~CreateAndDestroy()

```

```
32 { cout << "Object " << data << " destructor " << endl; }
```

图 6.9 调用构造函数和析构函数的顺序——create.cpp

```
33 //Fig. 6.9: fig06_09.cpp
34 //Demonstrating the order in which constructors and
35 //destructors are called.
36 #include <iostream>
37
38 using std::cout;
39 using std::endl;
40
41 #include "create.h"
42
43 void create( void ); //prototype
44
45 CreateAndDestroy first( 1 ); //global object
46
47 int main()
48 {
49     cout << " (global created before main)" << endl;
50
51     CreateAndDestroy second( 2 ); //local object
52     cout << " (local automatic in main)" << endl;
53
54     static CreateAndDestroy third( 3 ); //local object
55     cout << " (local static in main)" << endl;
56
57     create(); //call function to create objects
58
59     CreateAndDestroy fourth( 4 ); //local object
60     cout << " (local automatic in main)" << endl;
61     return 0;
62 }
63
64 //Function to create objects
65 void create( void )
66 {
67     CreateAndDestroy fifth( 5 );
68     cout << " (local automatic in create)" << endl;
69
70     static CreateAndDestroy sixth( 6 );
71     cout << " (local static in create)" << endl;
72
73     CreateAndDestroy seventh( 7 );
74     cout << " (local automatic in create)" << endl;
75 }
```

输出结果:

```
Object 1  constructor  (global created before main)
Object 2  constructor  (local automatic in main)
Object 3  constructor  (local static in main)
```

```

Object 5  constructor (local automatic in create)
Object 6  constructor (local static in create)
Object 7  constructor (local automatic in create)
Object 7  destructor
Object 5  destructor
Object 4  constructor (local automatic in main)
Object 4  destructor
Object 2  destructor
Object 6  destructor
Object 3  destructor
Object 1  destructor

```

图 6.9 调用构造函数和析构函数的顺序——fig06\_09.cpp

函数 create 声明了 3 个对象, fifth 和 seventh 是局部自动对象, sixth 是静态局部对象。程序执行到 create 的结尾时, 依次调用对象 seventh 和 fifth 的析构函数。由于对象 sixth 是静态局部对象, 因此直到程序结束时才退出。sixth 的析构函数在程序中止时删除所有其他对象之后和调用 third 和 first 的析构函数之前调用。

## 6.14 使用数据成员和成员函数

类的 private 数据成员只能由类的成员函数(和友元函数)访问。典型的操作包括由成员函数 computeInterest 调整客户的银行的借贷(即类 BankAccount 的 private 数据成员)。

类经常提供 public 成员函数, 让类的客户设置(写入)或获得(读取) private 数据成员的值。这些函数无需特别说明是 set 和 get 函数, 但事实却相反。更具体地说, 设置数据成员 interestRate 的成员函数通常命名为 setInterestRate, 获得 interestRate 的成员函数通常称为 getInterestRate。set 函数通常也称为“查询”函数。

提供 set 和 get 函数与指定数据成员为 public 同样重要, 这是 C++ 语言在软件工程中的另一个优势。如果数据成员是 public, 程序中的任何函数都可以随意读取或写入这个数据成员。如果数据成员是 private, 则 public get 函数可以让其他函数随意读取数据, 但是 get 函数控制数据的格式化和显示。而 public set 函数通常用于检查数据成员值的修改, 保证新值适合数据项目。例如, 把一个月的日期设置为 37 是不允许的, 将人的体重设置为负值也是不允许的, 将数字量设置为字母值也是非法的, 将一个人的测验成绩设置为 185(百分制成绩制)也是不允许的。

**软件工程知识 6.22** 指定 private 数据成员, 并通过 public 成员函数控制对这些数据成员的访问, 特别是写入访问, 可以保证数据的完整性。

**测试和调试提示 6.5** 指定 private 数据成员并不能自动保证数据的完整性, 程序员必须提供有效的检测。但是 C++ 提供了一个框架, 更有利于程序员用方便的方法设计更好的程序的框架。

**良好编程习惯 6.7** 设置 private 数据值的成员函数须验证所要的新值是否正确, 如果不正确, set 函数就将 private 数据成员设置为适当的一致状态。

试图将数据成员指定为无效值时,应提醒类的客户。类的 set 函数经常写为返回一个值,表示试图将数据成员指定为无效值。这使类的客户可以测试 set 函数的返回值,确定其操作对象是否为有效对象,如果对象是无效对象,则采取相应的操作。

图 6.10 中的程序将 Time 类扩展为包括 private 数据成员 hour, minute 和 second 的设置和 get 函数。set 函数严格控制数据成员的设置。如果要把任何数据成员设置为无效值,则会把数据成员设置为 0(因此使数据成员保持一致状态)。每个 get 函数只返回相应的数据成员的值。程序首先用 set 函数将 Time 对象 t 的 private 数据成员设置为有效值,接着用 get 函数找回该值以便输出。然后 set 函数将 hour 和 second 成员设置为无效值,将 minute 成员设置为有效值,接着用 get 函数找回该值以便输出。输出表明,无效值使得数据成员设置为 0。最后,程序将时间设置为 11:58:00,并且用函数 incrementMinutes 增加 3 分钟。函数 incrementMinutes 是个非成员函数,它调用设置和 get 函数增加 minute 成员的值。尽管这种方法有效,但是频繁的函数调用降低了程序的性能。下一章将介绍用友元函数消除多次函数调用引起的开销。

```

1 //Fig. 6.10: time3.h
2 //Declaration of the Time class.
3 //Member functions defined in time3.cpp
4
5 //preprocessor directives that
6 //prevent multiple inclusions of header file
7 #ifndef TIME3_H
8 #define TIME3_H
9
10 class Time {
11 public:
12     Time( int = 0, int = 0, int = 0 ); //constructor
13
14     //set functions
15     void setTime( int, int, int );      //set hour, minute, second
16     void setHour( int );                //set hour
17     void setMinute( int );              //set minute
18     void setSecond( int );              //set second
19
20     //get functions
21     int getHour();                      //return hour
22     int getMinute();                    //return minute
23     int getSecond();                    //return second
24
25     void printMilitary();                //output military time
26     void printStandard();                //output standard time
27
28 private:
29     int hour;                           //0 - 23
30     int minute;                         //0 - 59
31     int second;                         //0 - 59
32 };
33

```

34 #endif

图 6.10 使用 set 和 get 函数——tim3.h

```

35 //Fig. 6.10: time3.cpp
36 //Member function definitions for Time class.
37 #include <iostream>
38
39 using std::cout;
40
41 #include "time3.h"
42
43 //Constructor function to initialize private data.
44 //Calls member function setTime to set variables.
45 //Default values are 0 (see class definition).
46 Time::Time( int hr, int min, int sec )
47 { setTime( hr, min, sec ); }
48
49 //set the values of hour, minute, and second.
50 void Time::setTime( int h, int m, int s )
51 {
52     setHour( h );
53     setMinute( m );
54     setSecond( s );
55 }
56
57 //set the hour value
58 void Time::setHour( int h )
59 { hour = ( h >= 0 && h < 24 ) ? h : 0; }
60
61 //set the minute value
62 void Time::setMinute( int m )
63 { minute = ( m >= 0 && m < 60 ) ? m : 0; }
64
65 //set the second value
66 void Time::setSecond( int s )
67 { second = ( s >= 0 && s < 60 ) ? s : 0; }
68
69 //get the hour value
70 int Time::getHour() { return hour; }
71
72 //get the minute value
73 int Time::getMinute() { return minute; }
74
75 //get the second value
76 int Time::getSecond() { return second; }
77
78 //Print time in military format
79 void Time::printMilitary()
80 {
81     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
82           << ( minute < 10 ? "0" : "" ) << minute;

```

```

83 |
84
85 //Print time in standard format
86 void Time::printStandard()
87 |
88     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
89         << ":" << ( minute < 10 ? "0" : "" ) << minute
90         << ":" << ( second < 10 ? "0" : "" ) << second
91         << ( hour < 12 ? " AM" : " PM" );
92 |

```

图 6.10 使用 set 和 get 函数——tim3.cpp

```

93 //Fig. 6.10; fig06_10.cpp
94 //Demonstrating the Time class set and get functions
95 #include <iostream>
96
97 using std::cout;
98 using std::endl;
99
100 #include "time3.h"
101
102 void incrementMinutes( Time &, const int );
103
104 int main()
105 {
106     Time t;
107
108     t.setHour( 17 );
109     t.setMinute( 34 );
110     t.setSecond( 25 );
111
112     cout << "Result of setting all valid values:\n"
113         << " Hour: " << t.getHour()
114         << " Minute: " << t.getMinute()
115         << " Second: " << t.getSecond();
116
117     t.setHour( 234 ); //invalid hour set to 0
118     t.setMinute( 43 );
119     t.setSecond( 6373 ); //invalid second set to 0
120
121     cout << "\n\nResult of attempting to set invalid hour and"
122         << " second:\n Hour: " << t.getHour()
123         << " Minute: " << t.getMinute()
124         << " Second: " << t.getSecond() << "\n\n";
125
126     t.setTime( 11, 58, 0 );
127     incrementMinutes( t, 3 );
128
129     return 0;
130 }
131

```

```
132 void incrementMinutes(Time &tt, const int count)
133 {
134     cout << "Incrementing minute " << count
135         << " times;\nStart time: ";
136     tt.printStandard();
137
138     for ( int i = 0; i < count; i ++ ) {
139         tt.setMinute( ( tt.getMinute() + 1 ) % 60);
140
141         if ( tt.getMinute() == 0 )
142             tt.setHour( ( tt.getHour() + 1 ) % 24);
143
144         cout << "\nminute + 1: ";
145         tt.printStandard();
146     }
147
148     cout << endl;
149 }
```

输出结果:

Result of setting all valid values;

Hour:17 Minute:34 Second:25

Result of attempting to set invalid hour and second;

Hour:0 Minute:43 Second:0

Incrementing minute 3 times;

Start time: 11:58:00 AM

minute +1 :11:59:00 AM

minute +1: 12:00:00 PM

minute +1: 12:01:00 PM

图 6.10 使用 set 和 get 函数——fig06\_10.cpp

**常见编程错误 6.11** 构造函数可以调用类的其他成员函数,如 set 和 get 函数。但是因为构造函数初始化对象,所以数据成员可能还处于一致状态。数据成员未经初始化就开始使用可能造成逻辑错误。

从软件工程的角度看,使用 set 函数非常重要,因为它们可以执行有效性检查。在软件工程中,set 和 get 函数还有其他重要的优势。

**软件工程知识 6.23** 通过 set 和 get 函数访问 private 数据不仅能防止数据成员接收无效值,还能使类的客户忽略数据成员的表达方式。因此,如果数据成员的表达方式因为某种原因而改变(通常为了减少所需存储量或提高性能),只要成员函数提供的接口不变,那么需要改变的就只有成员函数,客户代码无须改变,但可能需要重新编译。

## 6.15 微妙的陷阱:返回对 private 数据成员的引用

对象的引用是对象名的别名,因此可以放在赋值语句左边。在这种情况下,引用可以作

为可接收赋值的左值。要使用这种功能,就要让类的 public 成员函数返回对该类 private 数据成员的非常量引用。

图 6.11 中的程序用简化的 Time 类演示了如何返回 private 数据成员的引用。调用 badsetHour 函数返回的引用作为 private 数据成员 hour 别名。函数调用可以像使用 private 数据成员一样,包括作为赋值语句的左值。

```

1 //Fig. 6.11: time4.h
2 //Declaration of the Time class.
3 //Member functions defined in time4.cpp
4
5 //preprocessor directives that
6 //prevent multiple inclusions of header file
7 #ifndef TIME4_H
8 #define TIME4_H
9
10 class Time {
11 public:
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int, int, int );
14     int getHour();
15     int &badsetHour( int ); //DANGEROUS reference return
16 private:
17     int hour;
18     int minute;
19     int second;
20 };
21
22 #endif

```

图 6.11 返回对 private 数据成员的引用——time4. h

```

23 //Fig. 6.11: time4.cpp
24 //Member function definitions for Time class.
25 #include "time4.h"
26
27 //Constructor function to initialize private data.
28 //Calls member function setTime to set variables.
29 //Default values are 0 (see class definition).
30 Time::Time( int hr, int min, int sec )
31 { setTime( hr, min, sec ); }
32
33 //set the values of hour, minute, and second.
34 void Time::setTime( int h, int m, int s )
35 {
36     hour = ( h >= 0 && h < 24 ) ? h : 0;
37     minute = ( m >= 0 && m < 60 ) ? m : 0;
38     second = ( s >= 0 && s < 60 ) ? s : 0;
39 }
40
41 //get the hour value

```



```

42 int Time::getHour() { return hour; }
43
44 //POOR PROGRAMMING PRACTICE;
45 //Returning a reference to a private data member.
46 int &Time::badsetHour( int hh )
47 {
48     hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
49
50     return hour; //DANGEROUS reference return
51 }

```

图 6.11 返回对 private 数据成员的引用——time4. cpp

```

52 //Fig. 6.11: fig06_11.cpp
53 //Demonstrating a public member function that
54 //returns a reference to a private data member.
55 //Time class has been trimmed for this example.
56 #include <iostream>
57
58 using std::cout;
59 using std::endl;
60
61 #include "time4.h"
62
63 int main()
64 {
65     Time t;
66     int &hourRef = t.badsetHour( 20 );
67
68     cout << "Hour before modification: " << hourRef;
69     hourRef = 30; //modification with invalid value
70     cout << "\nHour after modification: " << t.getHour();
71
72     //Dangerous: Function call that returns
73     //a reference can be used as an lvalue!
74     t.badsetHour(12) = 74;
75     cout << "\n\n*****\n\n"
76         << "POOR PROGRAMMING PRACTICE!!!!!! \n"
77         << "badsetHour as an lvalue, Hour: "
78         << t.getHour()
79         << "\n*****\n\n" << endl;
80
81     return 0;
82 }

```

输出结果:

```

Hour before modification:20
Hour after modification:30

```

```

*****
POOR PROGRAMMING PRACTICE!!!!!!

```

```
badSetHour as an lvalue,Hour;74
*****
```

图 6.11 返回对 private 数据成员的引用——fig06\_11.cpp

**良好编程习惯 6.8** 绝不要让类的 public 成员函数返回对该类 private 数据成员的非常量引用(或指针)。返回这种引用会破坏类的封装。实际上,返回任何对 private 数据的引用或指针仍然使客户代码通过类的数据的表达式来决定。因此,应该避免返回对 private 数据的指针或引用。

程序首先声明 Time 对象 t 和引用 hourRef, 该引用是调用 t.badSetHour(20) 返回的引用赋给的。程序显示别名 hourRef 的值。然后用这个别名设置 hour 的值为 30(无效值)并再次显示该值。最后,函数调用本身被用作左值并赋值为 74(另一个无效值),再次显示该值。

## 6.16 通过默认的按位成员复制赋值

赋值操作符(=)可以将一个对象赋给另一个类型相同的对象。这种赋值默认通过成员复制进行,对象的每个成员逐一复制(赋值)给另一对象的同一成员(如图 6.12 所示)。注意,如果类的数据成员包含动态分配内存的对象,那么通过默认的按位成员复制赋值可能会引起严重问题,第 8 章将介绍这些问题及其解决办法。

对象可以作为函数参数进行传递并从函数返回。这种传递和返回默认按传值调用进行,即传递和返回对象的副本(详情参见第 8 章)。

**性能提示 6.4** 传值调用传递对象的安全性较高,因为被调函数无法访问原始对象,但如果要复制大对象,传值调用则可能降低性能。对象按引用调用传递时可以按指针或对象引用传递。按引用调用有性能优势,但安全性较低,因为被调函数可以访问原始对象。按常量引用调用则既安全,又有性能上的优势。

```
1 //Fig. 6.12: fig06_12.cpp
2 //Demonstrating that class objects can be assigned
3 //to each other using default memberwise copy
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 //Simple Date class
10 class Date {
11 public:
12     Date( int = 1, int = 1, int = 1990 ); //default constructor
13     void print();
14 private:
15     int month;
16     int day;
17     int year;
```

```
18 };
19
20 //Simple Date constructor with no range checking
21 Date::Date( int m, int d, int y )
22 {
23     month = m;
24     day = d;
25     year = y;
26 }
27 28 //Print the Date in the form mm-dd-yyyy
29 void Date::print()
30 { cout << month << '-' << day << '-' << year; }
31
32 int main()
33 {
34     Date date1( 7, 4, 1993 ), date2; //d2 defaults to 1/1/90
35
36     cout << "date1 = ";
37     date1.print();
38     cout << "\ndate2 = ";
39     date2.print();
40
41     date2 = date1; //assignment by default memberwise copy
42     cout << "\n\nAfter default memberwise copy, date2 = ";
43     date2.print();
44     cout << endl;
45
46     return 0;
47 }
```

输出结果:

```
date1 = 7-4-1993
date2 = 1-1-1990
```

```
After default memberwise copy, date2 = 7-4-1993
```

图 6.12 通过默认的按位成员复制赋值

## 6.17 软件重用性

编写面向对象程序时,人们将重点放在实现有用的类。类的获取和分类非常容易,所以程序员可以很方便地使用它们。许多类库已经存在,许多类库还在不断开发中。人们正不断地推广这些类库的使用。软件越来越趋向于通过现有的、定义良好、经过认真测试、文档齐全、可移植的各种组件进行构建。这种软件重用性加速了功能强大的、高质量软件的开发速度。通过重用组件实现快速应用程序开发(RAD)已是大势所趋。

然而,在发挥软件重用性的巨大潜力之前,还有几个重大问题有待解决。我们需要有分类机制、许可证机制,用保护机制来区分类的主副本,用描述机制让新系统设计人员确定现

有对象是否可满足需求,用浏览机制确定有什么类最接近软件开发人员的需求。许多有趣的研究和开发问题急需解决。人们正在积极解决这些问题,因为这种方案具有强大的潜在价值。

## 6.18 【可选案例分析】对象思想:编写电梯模拟程序所需的类

第1章到第5章的“对象思想”小节介绍了面向对象的基础知识,并开始了电梯模拟程序的面向对象设计。第6章介绍了编程和使用C++类的细节。接下来将用UML类图表编写定义类的头文件。

### 6.18.1 实现方法:可见性

第6章介绍了访问说明符 `public` 和 `private`。在生成类的头文件之前,必须先考虑类图表中哪些元素是 `public` 类型,哪些元素是 `private` 类型。

**软件工程知识 6.24** 在证明类元素需要 `public` 可见性之前,每个类元素需具有 `private` 可见性。

第6章介绍了数据成员一般应如何指定为 `private`,那么成员函数如何指定呢?类的操作就是类的成员函数。这些操作必须用该类的客户代码调用,所以,成员函数应为 `public` 类型。在UML中,在特定元素(即成员函数或数据成员)前加一个加号(+)即可表示 `public` 可见性;加一个减号(-)表示 `private` 可见性。图6.13中的程序提供了一个包含可见性符号的更新类图表。注意,图4.27中的序列图表中增加了对类 `Floor` 的 `personArrives` 操作。正如在系统中类的C++头文件中编写的,自动将带有“+”的项放入人类声明的 `public`,将带有“-”的项放入 `private`。

### 6.18.2 实现方法:句柄

要让A类的对象与B类的对象沟通,A类的对象必须有一个连接B类的对象的 `handle`(句柄)。这意味着A类的对象必须知道B类的对象的名称,或者A类对象必须有B类对象的引用(参见3.17节)或指针(参见第5章)。<sup>①</sup>图5.36提供了一个系统中类之间的合作列表。表中左列的类需要包含指向右列每个类的句柄,以向这个类发送信息。图6.14以图5.16展示的信息为基础,列出了每个类的句柄。

第6章介绍了C++中如何将句柄作为类的引用和指针执行(引用比指针更常用)。然后这些引用变成类的属性(数据)。我们打算在此逐一介绍图6.14中类的头文件中的各项,而是做一个简要介绍,详情将在第7章描述。

<sup>①</sup> B类对象的名称不适用于A类对象时,我们会优先选用引用,而不是指针(在适当的时候),因为引用原本比指针更安全。

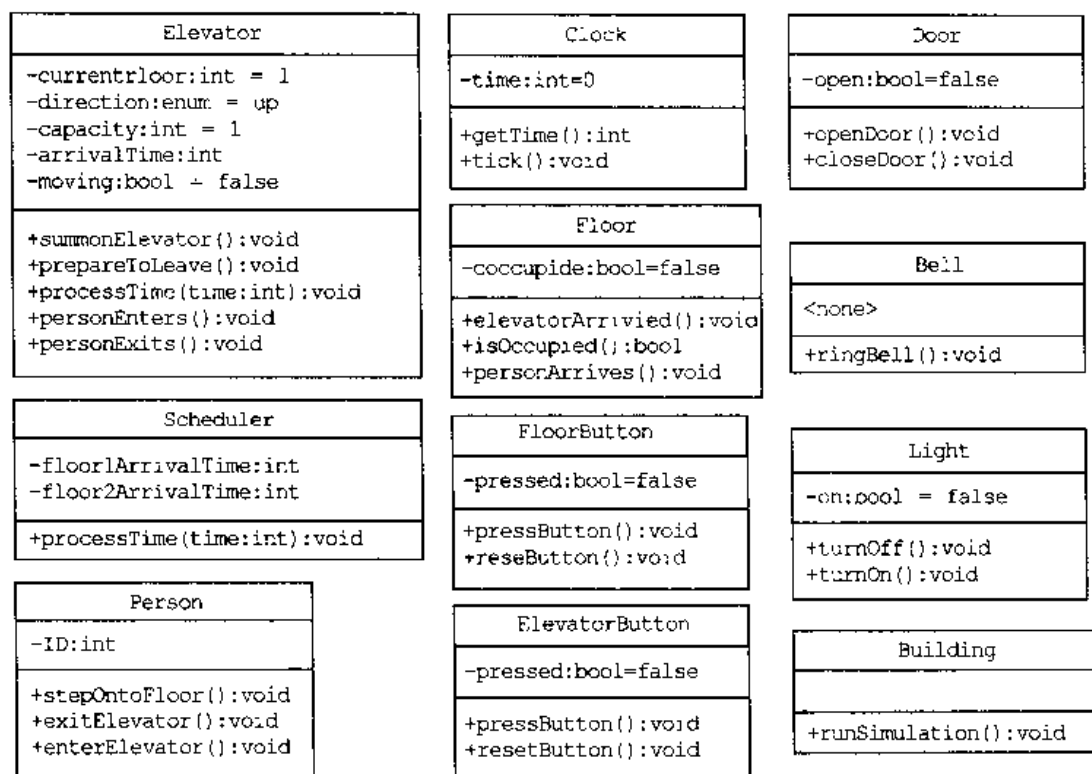


图 6.13 用可见性符号完成类图表

| 类              | 句柄                                           |
|----------------|----------------------------------------------|
| Elevator       | ElevatorButton, Bell, Floor, Door            |
| Clock          |                                              |
| Scheduler      | Person, Floor                                |
| Person         | FloorButton, ElevatorButton, Elevator, Floor |
| Floor          | FloorButton, Light                           |
| FloorButton    | Elevator                                     |
| ElevatorButton | Elevator                                     |
| Door           | Person                                       |
| Bell           |                                              |
| Light          |                                              |
| Building       | Clock, Scheduler, Elevator                   |

图 6.14 各类的句柄列表

### 6.18.3 实现方法:类的头文件

前面已经介绍了设计C++程序中的类,接下来开始编写电梯模拟程序的代码。这一节将测试系统中每一个类的头文件。第7章的“对象思想”中,介绍了模拟程序中完整的操作C++代码。第9章将修改代码,集成继承。

为演示构造函数和析构函数运行的顺序,要对每个类的构造函数和析构函数进行编码,从而显示这些函数正在运行的信息。在头文件包含了构造函数和析构函数的原型,第7章

介绍的 .cpp 文件包含了这些函数的实现方法。

图 6.15 中的程序列出了 Bell 类的头文件。根据类图表的操作(如图 6.13 所示),声明构造函数和析构函数(第 8 行和第 9 行)以及成员函数 ringBell(第 10 行),每个成员函数都具有 public 可见性。由于对该类未定义其他 public 或 private 元素,因此头文件是完整的。

```

1 //bell.h
2 //Definition for class Bell.
3 #ifndef BELL_H
4 #define BELL_H
5
6 class Bell {
7 public:
8     Bell();           //constructor
9     ~Bell();          //destructor
10    void ringBell() const; //ring the bell
11 };
12
13 #endif //BELL_H

```

图 6.15 Bell 类的头文件

图 6.16 列出了 Clock 类的头文件。包含图 6.13 中的构造函数和析构函数(第 8 行和第 9 行)以及 public 成员函数 tick() 和 getTime()(第 10 行和第 11 行)。通过声明 int 类型的 private 数据成员 time(第 13 行)实现头文件中的 time 属性。模拟程序中每增加一秒,Building 类的对象调用 Clock 类对象的 getTime 成员函数获得 time 的当前值,调用 tick 成员函数增加 time。

```

1 //clock.h
2 //Definition for class Clock.
3 #ifndef CLOCK_H
4 #define CLOCK_H
5
6 class Clock {
7 public:
8     Clock();           //constructor
9     ~Clock();          //destructor
10    void tick();         //increment clock by one second
11    int getTime() const; //returns clock's current time
12 private:
13    int time;           //clock's time
14 };
15
16 #endif //CLOCK_H

```

图 6.16 Clock 类的头文件

图 6.17 列出了 Person 类的头文件。声明类图表(图 6.13)中的 ID 属性(第 16 行)以及操作(第 12~14 行)setOntoFloor, enterElevator 和 exitElevator。同时还声明了返回乘客 ID 号

码的 public getID 成员函数(第 10 行),我们将用该操作跟踪模拟程序中的乘客。

```

1 //person.h
2 //definition of class Person
3 #ifndef PERSON_H
4 #define PERSON_H
5
6 class Floor;           //forward declaration
7 class Elevator;        //forward declaration
8
9 class Person {
10
11 public:
12     Person( const int );    //constructor
13     ~Person();             //destructor
14     int getID() const;      //returns person's ID
15
16     void stepOntoFloor( Floor & );
17     void enterElevator( Elevator &, Floor & );
18     void exitElevator( const Floor &, Elevator & ) const;
19
20 private:
21     static int personCount; //total number of persons
22     const int ID;           //person's unique ID #
23     const int destinationFloor; //destination floor #
24 };
25
26 #endif //PERSON_H

```

图 6.17 Person 类的头文件

模拟程序的开始位置不生成 Person 类的对象,而是在模拟程序运行时随机动态生成。基于此,实现 Person 类对象要不同于实现系统中其他类对象。第 7 章介绍如何动态地生成新对象之后,将为 Person 类的头文件增加重要元素。

图 6.18 列出 Door 类的头文件。声明构造函数和析构函数(第 8 行和第 9 行)以及 public 成员函数 openDoor 和 closeDoor(第 11 行和 12 行)。同时还声明了 private 类数据成员 open(第 14 行)。图 6.14 中的图表说明 Door 类需要一个 Person 类的句柄。但是,因为 Person 类对象是在系统中动态地生成的,因此在这一点上无法确定如何实现到 Person 类对象的句柄。第 7 章介绍完动态地生成对象之后,我们将有更好的办法实现到 Person 类的句柄。

```

1 //door.h
2 //Definition for class Door.
3 #ifndef DOOR_H
4 #define DOOR_H
5
6 class Person;           //forward declaration
7 class Floor;           //forward declaration
8 class Elevator;         //forward declaration
9

```

```

10 class Door {
11
12 public:
13     Door();           //constructor
14     ~Door();          //destructor
15
16     void openDoor( Person * const, Person * const,
17                   Floor &, Elevator & );
18     void closeDoor( const Floor & );
19
20 private:
21     bool open;         //open or closed
22 };
23
24 #endif //DOOR_H

```

图 6.18 Door 类的头文件

图 6.19 列出 Light 类的头文件。图 6.13 中的图表提供的信息可以引导大家声明 public 成员函数 turnOn, turnOff 以及 bool 类型的 private 数据成员 on。这个头文件还介绍了实现方法的新内容——区分系统中同一类的不同对象。模拟程序包含两个 Light 类的对象：一个对象属于第一楼层，一个对象属于第二楼层。由于输出时无法区分这两个对象，所以必须给每一个对象不同的名字。所以把第 14 行

```
char *name; //which floor the Light is on
```

添入类声明的 private。在构造函数中添加参数 char \* (第 8 行)，可以将 Light 类的每一个对象名初始化。

```

1 //light.h
2 //Definition for class Light.
3 #ifndef LIGHT_H
4 #define LIGHT_H
5
6 class Light {
7 public:
8     Light( const char * );   //constructor
9     ~Light();               //destructor
10    void turnOn();           //turns light on
11    void turnOff();          //turns light off
12 private:
13    bool on;                 //true if on; false if off
14    const char *name;        //which floor the light is on
15 };
16
17 #endif //LIGHT_H

```

图 6.19 Light 类的头文件

图 6.20 列出了 Building 类的头文件。

```

1 //building.h
2 //Definition for class Building.

```



```

3  #ifndef BUILDING_H
4  #define BUILDING_H
5
6  #include "elevator.h"
7  #include "floor.h"
8  #include "clock.h"
9  #include "scheduler.h"
10
11 class Building {
12
13 public:
14     Building();           //constructor
15     ~Building();          //destructor
16     void runSimulation( int ); //run simulation for specified time
17
18 private:
19     Floor floor1;         //floor1 object
20     Floor floor2;         //floor2 object
21     Elevator elevator;    //elevator object
22     Clock clock;          //clock object
23     Scheduler scheduler;  //scheduler object
24 };
25
26 #endif //BUILDING_H

```

图 6.20 Building 类的头文件

类声明的 public 部分包含图 6.13 中的构造函数、析构函数和 runSimulation 成员函数。第 4 章里第一次定义 runSimulation 操作时,并不知道什么对象将调用函数,开始运行模拟程序。介绍了 C++ 中的类后,可以知道在 main 中需要声明 Building 类的对象,并且 main 将会调用 runSimulation。主程序中的代码为:

```

Building building;           //create the building object
Building.runSimulation();    //invoke runSimulation

```

我们还选择在 runSimulation 声明中包含一个 int 类型的参数。Building 对象将运行电梯模拟程序,通过该参数将秒数传递给对象。上述 runSimulation 调用将包含一个表示模拟程序持续时间的数字。图 6.14 表示 Building 类需要其组成对象的句柄。此时还不能实现这些句柄,因为尚未介绍如何组成。第 7 章之前,还不能介绍 Building 类的组成对象的实现方法(见图 6.20 的第 14~18 行)。

图 6.21 列出了 ElevatorButton 类的头文件。我们通过图 6.13 中的类图象声明了 pressed 属性、pressButton 和 resetButton 成员函数及其构造函数和析构函数。图 6.14 表明 ElevatorButton 类需要一个电梯句柄。第 19 行

```
Elevator &elevatorRef;
```

包含这个句柄(注意,我们使用了一个用以实现句柄的引用)。第 7 章将介绍如何使用这个引用向电梯发送信息。

```

1  //elevatorButton.h
2  //Definition for class ElevatorButton.
3  #ifndef ELEVATORBUTTON_H

```

```

4 #define ELEVATORBUTTON_H
5
6 class Elevator;           //forward declaration
7
8 class ElevatorButton {
9
10 public:
11     ElevatorButton( Elevator & );   //constructor
12     ~ElevatorButton();             //destructor
13
14     void pressButton();             //press the button
15     void resetButton();            //reset the button
16
17 private:
18     bool pressed;                  //state of button
19     Elevator &elevatorRef;          //reference to button's elevator
20 };
21
22 #endif //ELEVATORBUTTON_H

```

图 6.21 ElevatorButton 类的头文件

声明一个引用时,引用必须初始化,但是不允许将数值赋给头文件中的类数据成员。因此,引用必须在构造函数中初始化,第10行将 Elevator 引用作为一个参数传递到构造函数。

#### 第6行

```
class Elevator; //forward declaration
```

是 Elevator 类的提前声明。提前声明允许声明一个 Elevator 类对象引用,而不需要在 ElevatorButton 类的头文件中包含 Elevator 类的头文件<sup>①</sup>。

图 6.22 列出 FloorButton 类的头文件。这个头文件和 ElevatorButton 类的头文件相同,除非声明 int 类型的 private 数据成员 floorNumber。为获得模拟程序输出, FloorButton 类的对象需要知道自己属于哪一个楼层。楼层号作为构造函数参数进行传递以实现初始化(第10行)。

```

1 //floorButton.h
2 //Definition for class FloorButton.
3 #ifndef FLOORBUTTON_H
4 #define FLOORBUTTON_H
5
6 class Elevator;           //forward declaration
7
8 class FloorButton {
9 public:
10     FloorButton( const int, Elevator & );   //constructor
11     ~FloorButton();             //destructor
12
13     void pressButton();             //press the button

```

<sup>①</sup> 如有可能,尽量利用提前声明(在可能的情形下),而不是将整个头文件包含在内,有助于避免名为“循环包容”的预处理程序问题。第7章将详细讨论循环包容问题。

```

14 void resetButton();           //reset the button
15 private:
16     const int floorNumber;     //number of the button's floor
17     bool pressed;              //state of button
18
19 //reference to button's elevator
20 Elevator &elevatorRef;
21 };
22
23 #endif //FLOORBUTTON_H

```

图 6.22 FloorButton 类的头文件

图 6.23 列出了 Scheduler 类的头文件,第 23 行和第 24 行

```

int floor1ArrivalTime;
int floor2ArrivalTime;

```

声明了 Scheduler 类的 private 数据成员,其属性与图 6.13 定义的该类(图 6.13)的属性相同。第 12 行声明了 public 成员函数 processTime,其操作与第 4 章“对象思想”中定义的操作相同。

```

1 //scheduler.h
2 //defintion for class Scheduler
3 #ifndef SCHEDULER_H
4 #define SCHEDULER_H
5
6 class Floor;           //forward declaration
7
8 class Scheduler {
9 public:
10     Scheduler( Floor &, Floor & ) //constructor
11     ~Scheduler();              //destructor
12     void processTime( int );    //set scheduler's time
13 private:
14
15     //method that schedules arrivals to a specified floor
16     void scheduleTime( Floor & );
17
18     //method that delays arrivals to a specified floor
19     void delayTime( Floor & );
20
21     Floor &floor1Ref;
22     Floor &floor2Ref;
23     int floor1ArrivalTime;
24     int floor2ArrivalTime;
25 }
26
27 #endif //SCHEDULER_H

```

图 6.23 Scheduler 类的头文件

第 15~19 行表明图 4.27 序列图表中定义的函数。每个函数取一个 Floor 类对象的引

用作为参数。注意,我们没有将这些函数作为操作(如 public 成员函数)列出,因为客户对象不调用这些方法,只有 Scheduler 类使用这些方法执行其内部操作。因此将这些方法放入类声明的 private 部分。

第 21 行和第 22 行声明图 6.14 定义的句柄,还将每个句柄实现为 Floor 类对象的引用。Scheduler 类需要这些句柄将 isOccupied 信息发送到模拟程序(参见图 4.27)中的两个楼层。还必须要在第 6 行中做一个 Floor 类的提前声明,以便声明引用。

图 6.24 包含了 Floor 类的头文件,声明图 6.13 中的 public 成员函数 elevatorArrived, isOccupied 和 personArrives。第 26 行声明了 public 成员函数 elevatorLeaving。添加该函数可使电梯可以通知楼层“电梯准备离开”。电梯调用 elevatorLeaving 操作,楼层关掉本层信号灯。

```

1 //floor.h
2 //Definition for class Floor.
3 #ifndef FLOOR_H
4 #define FLOOR_H
5
6 class Elevator;           // forward declaration
7
8 class Floor {
9 public:
10     Floor( int, Elevator & );    // constructor
11     ~Floor();                  // destructor
12
13     //return true if floor is occupied
14     bool isOccupied();
15
16     //return floor's number
17     int getNumber();
18
19     //pass a handle to new person coming on floor
20     void personArrives();
21
22     //notify floor that elevator has arrived
23     void elevatorArrived();
24
25     //notify floor that elevator is leaving
26     void elevatorLeaving();
27
28     //declaration of FloorButton component ( see Chapter 7 )
29
30 private:
31     int floorNumber;           //the floor's number
32     Elevator &elevatorRef;      //pointer to elevator
33     //declaration of Light component ( see Chapter 7 )
34 }
35
36 #endif //FLOOR_H

```

图 6.24 Floor 类的头文件

第 31 行中添加了 `private` 类型的 `floorNumber` 数据成员, 添加该数值的目的同样为了输出, 如同我们添加的类 `FloorButton` 的 `floorNumber` 数据成员一样。将一个 `int` 类型的参数添加到构造函数, 使构造函数可以初始化这个数据成员。还声明了到图 6. 14 中定义的 `Elevator` 类的句柄。Floor 类的组成成员声明(参见第 28 行)将在第 7 章介绍。

图 6. 25 列出了 `Elevator` 类的头文件。头文件的 `public` 部分中, 声明了图 6. 13 中列出的 `summonElevator`, `prepareToLeave` 和 `processTime` 操作。要区分等候电梯的人和电梯里的人需要重新命名类 `Elevator` 列出的最后两个操作。我们将这两个操称为 `passengerEnters` 和 `passengerExits`, 并在头文件的 `public` 部分声明它们。此外还声明了每个楼层的引用(第 38 至第 39 行), 第 10 行的构造函数将初始化这些引用。

```

1 //elevator.h
2 //Definition for class Elevator.
3 #ifndef ELEVATOR_H
4 #define ELEVATOR_H
5
6 class Floor;           // forward declaration
7
8 class Elevator {
9 public:
10     Elevator( Floor &,Floor & );    //constructor
11     ~Elevator();                 //destructor
12
13     //request that elevator service a particluar floor
14     void summonElevator( int );
15
16     //prepare elevator to leave
17     void prepareToLeave();
18
19     //give time to elevator
20     void processTime( int );
21
22     //notify elevator that passenger is boarding
23     void passengerEnters();
24
25     //notify elevator that passenger is exiting
26     void passengerExits();
27
28     //declaration of ElevatorButton component( see Chapter 7)
29 private:
30     bool moving;                //elevator state
31     int direction;              //current direction
32     int currentFloor;           //current location
33
34     //time for arrival at a floor
35     int arrivalTime;
36
37     //References to floors serviced by elevator
38     Floor &floor1Ref;

```

```

39   Floor &floor2Ref;
40
41   //declaration of Door component( see Chapter 7 )
42   //declaration of Bell component ( see Chapter 7 )
43   |
44
45 #endif //ELEVATOR_H

```

图 6.25 Elevator 类的头文件

头文件的 private 部分声明了图 6.13 中的 moving、direction、currentFloor 和 arrivalTime 属性。不要求声明 capacity 属性,但应写出代码,保证每次只有一人在电梯里。

#### 6.18.4 结论

下一章的“对象思想”小节将介绍完整的电梯模拟程序的C++代码。使用下一章介绍的概念实现组成关系、类 Person 对象的动态生成以及 static 和 const 数据成员和函数。第9章的“对象思想”小节,将利用继承进一步完善面向对象电梯模拟程序的设计和实现方法。

### 6.19 小结

- 结构是用其他类型元素建立的混合数据类型。
- 结构定义用关键字 struct 开始,结构体放在一对花括号( { } )中。每个结构定义都必须用分号结束。
- 结构标志名可用于声明结构类型的变量。
- 结构定义没有在内存中保留任何空间;而是生成新的数据类型,用于声明变量。
- 使用成员访问操作符,即圆点操作符( . )和箭头操作符( -> )访问结构成员或类成员。圆点操作符通过对象的变量名或对象的引用访问结构和类成员。箭头操作符通过对象指针访问结构和类成员。
- 通过 struct 生成新数据类型有一定的缺陷,会出现未初始化的数据。如果 struct 的实现方法改变,使用该 struct 的所有程序都要改变。没有保护机制可保证数据的正确和数据的一致性。
- 类使程序员可以构造有属性和行为的对象。C++ 中用关键字 class 或 struct 定义类的类型。通常用关键字 class。
- 可以用类名声明该类的对象。
- 类定义以关键字 class 开始。类定义放在一对花括号( { } )之间。类定义用分号结束。
- 任何可以访问类的对象的函数可以访问在 public: 后面声明的数据成员和成员函数。
- private: 后面声明的所有数据成员和成员函数只能由该类的成员函数和友元访问。
- 成员访问说明符始终要加上冒号( : ),可以在类定义中以任何顺序多次出现。
- 不能在类的外部访问 private 数据。
- 类的实现方法对客户是隐藏的。
- 构造函数是特殊的成员函数,初始化类对象的数据成员。类的构造函数供生成这个

类的对象自动调用。

- 与类同名而且前面加上代字符( ~ )的函数称为类的析构函数。
- 类的 public 成员函数集称为类的接口或 public 接口。
- 在类定义外定义成员函数时,函数名前要加上类名和二元作用域分辨符(::)。
- 尽管类定义中声明的成员函数可以在类定义外定义,但成员函数仍然在类作用域中。
- 如果在类定义中定义成员函数,该成员函数将自动成为内联函数,但编译器有权决定其是否作为内联函数。
- 成员函数调用比过程式编程中的传统函数调用更简练,因为成员函数使用的大多数数据可以直接在对象中访问。
- 在类范围中,类成员可以简单地按名称引用。在类作用域之外,类成员通过对象名、对象引用和对象指针来引用。
- 成员选择操作符, 和 -> 用于访问类成员。
- 良好软件工程的基本原则之一是将接口与实现分离。
- 类定义通常放在头文件中,类的成员函数定义放在同一基本名称的源代码文件中。
- 类的默认访问模式是 private,所以类名和第一个说明符(例如 public)之间的所有成员都是 private 类型。
- 类的 public 成员主要用于向类的客户代码提供类的服务。
- 访问类的 private 数据应当由名为访问函数的成员函数进行控制。如果类允许客户代码读取 private 数据的值,可以提供一个 get 函数;如果类允许客户代码修改 private 数据的值,则可以提供一个 set 函数。
- 类的数据成员通常指定为 private,类的成员函数通常指定为 public。有些成员函数可以是 private,作为工具函数为类的其他函数服务。
- 类的数据成员不能在类定义中初始化,应在类的构造函数中初始化或在生成对象之后设置其数值。
- 可以重载构造函数。
- 类对象初始化之后,操作该对象的所有成员函数应保证对象处于一致状态。
- 声明类对象时可以提供初始化值,这些初始化值传递给类的构造函数。
- 构造函数可以指定默认参数。
- 构造函数不指定返回的类型,也不返回数值。
- 如果类不定义构造函数,编译器会生成默认构造函数。编译器生成的构造函数不执行任何初始化,因此生成对象时,不能保证一致状态。
- 自动对象的析构函数在对象离开作用域时调用,析构函数本身并不删除对象,而是进行系统释放对象内存之前的清理工作,使内存可以重用于保存新对象。
- 析构函数不接受参数也不返回数值。类只能有一个析构函数,而且析构函数不允许重载。
- 赋值操作符( = )可以将一个对象赋给另一个类型的对象,这种赋值方式一般通过默认的按位成员复制来完成。按位成员复制不适用于所有的类。

## 本章术语

- & reference operator 引用操作符
- abstract data type (ADT) 抽象数据类型
- access function 访问函数
- arrow member selection operator( -> )  
箭头成员选择操作符( -> )
- attribute 属性
- behavior 行为
- binary scope resolution operator( :: )  
二元作用域分辨符( :: )
- class definition 类定义
- class member selector operator( . )  
类成员选择操作符( . )
- class scope 类作用域
- client of class 类客户
- consistent state for a data member  
数据成员的一致状态
- constructor 构造函数
- data member 数据成员
- data type 数据类型
- default constructor 默认构造函数
- destructor 析构函数
- dot member selection operator( . )  
圆点成员选择操作符( . )
- encapsulation 封装
- extensibility 可扩展性
- file scope 文件范围
- get function get 函数
- global object 全局对象
- header file 头文件
- helper function 帮助函数
- implementation of a class 类的实现方法
- information hiding 信息隐藏
- initialize a class object 初始化类对象
- inline member function inline 成员函数
- instance of class 类的实例
- instantiate an object of a class 实例化类对象
- interface to a class 类的接口
- member access control 成员访问控制
- member access specifiers 成员访问说明符
- member function 成员函数
- member initializer 成员初始化值
- member selection operator( . and -> )  
成员选择操作符( . 和 -> )
- memberwise copy 按位成员复制
- message 信息
- member function 成员函数
- nonstatic local object 非静态局部对象
- object 对象
- object-oriented design( OOD ) 面向对象设计
- object-oriented programming( OOP ) 面向对象编程
- predicate function 判定函数
- principle of least privilege 最低权限原则
- procedural programming 过程式编程
- programmer-defined type 程序员定义类型
- proxy class 代理类
- public interface of a class 类的 public 接口
- query function 查询函数
- rapid applications development( RAD )  
快速应用程序开发
- reusable code 可复用代码
- scope resolution operator( :: ) 作用域分辨符( :: )
- self-referential structure 自引用结构
- services of a class 类服务
- set function set 函数
- software reusability 软件重用性
- source-code file 源代码文件
- static local object 静态局部对象
- structure 结构
- tilde ( ~ ) in destructor name  
析构函数名中的代字符( ~ )
- user-defined types 用户自定义类型
- utility function 工具函数

## “对象思想”术语

- “ + ” symbol for public visibility  
表示 public 可见性的符号“ + ”
- “ - ” symbol for private visibility  
表示 private 可见性的符号“ - ”
- circular include problem 循环包容问题
- forward declaration 提前声明
- handle 句柄
- private visibility private 可见性



public visibility public 可见性

visibility 可见性

references vs. pointers 引用比较指针

## 常见编程错误

- 6.1 表达式(\*timePtr).hour 指 timePtr 所指 struct 的 hour 成员。省略括号的 \*timePtr 是语法错误,因为“.”的优先级高于“\*”,所以表达式变成\*(timePtr.hour)。这是语法错误,因为指针必须要用箭头引用成员。
- 6.2 在类(结构)定义结尾处不加分号是语法错误。
- 6.3 对构造函数指定返回类型或返回值是语法错误。
- 6.4 试图将类定义中显式初始化类的数据成员是语法错误。
- 6.5 在类外部定义类的成员函数时,忽略函数名中的类名和作用域分辨符是错误。
- 6.6 不是特定类的成员函数(或类的友元)试图访问类的 private 成员是语法错误。
- 6.7 类的数据成员不能在类定义中初始化。
- 6.8 试图声明构造函数的返回类型或试图从构造函数中返回值是语法错误。
- 6.9 在头文件和成员函数定义中指定同一成员函数的默认初始化值。
- 6.10 向析构函数传递参数、指定析构函数的返回值类型(即使指定了 void)以及从析构函数返回数值或重载析构函数都是语法错误。
- 6.11 构造函数可以调用类的其他成员函数,如 set 和 get 函数。但是因为构造函数初始化对象,所以数据成员可能还处于一致的状态。数据成员未经初始化就开始使用可能造成逻辑错误。

## 良好编程习惯

- 6.1 为保证类定义的清晰性和可读性,每个成员访问说明符只在类定义中使用一次。public 成员会放在前面,以便于寻找。
- 6.2 头文件的 #ifndef 和 #define 预处理程序指令中使用头文件名,并用下划线(\_)代替圆点(.)。
- 6.3 如果可以选择在类定义中先列出 private 成员,尽管程序默认访问模式为 private,但最好显式使用 private 成员访问说明符,这样可使程序更清晰。我们提倡先列出类的 public 成员以强调类的接口。
- 6.4 尽管 public 和 private 成员访问说明符可以重复和混用,但最好先将所有类的 public 成员列成一组,然后再将所有类的 private 成员列成一组。这样可以使用户更注类的 public 接口,而不是类的实现方法。
- 6.5 适当的时候(几乎一直如此)就提供构造函数,保证各个对象正确初始化为有意义的值。针数据成员尤其如此,应初始化为某个合法指针值或初始化为 0。
- 6.6 只在头文件中类定义内部的函数原型中声明默认函数参数值。
- 6.7 设置 private 数据值的成员函数应验证所要的新值是否正确,如果不正确, set 函数应将 private 数据成员设置为适当的一致状态。
- 6.8 不要让类的 public 成员函数返回对该类 private 数据成员的非常量引用(或指针)。返回这种引用会破坏类的封装。实际上,返回任何对 private 数据引用或指针仍然使客户的

代码由类的数据的表达式决定。因此,应该避免返回对 private 数据的指针或引用。

### 性能提示

- 6.1 结构通常通过传值调用传递。要避免复制结构的开销,按引用调用传递结构。
- 6.2 在类定义中定义小的成员函数将自动使该成员函数成为内联函数(如果编译器如此选择的话)。这样可以提高性能,但是不能提高软件工程质量,因为类的客户可以看到函数的实现,并且如果内联函数发生了变化,就必须重新编译代码。
- 6.3 实际上,对象只包含数据,所以比其中包含函数的对象要小得多。对类名或该类的对象采用 sizeof 操作符时,只能得到类的数据的长度。编译器生成独立于所有类的对象的成员函数的副本(只有一份)。类的所有对象共享这个成员函数的唯一副本。当然,每个对象都需要自己的类数据副本,因为对象中的数据是不同的。函数代码是不可修改的(也称为可重入码或纯过程),因而可供类的所有对象共享。
- 6.4 传值调用传递对象的安全性较高,因为被调函数无法访问原始对象,但如果要复制大对象,传值调用则可能降低性能。对象按引用调用传递时可以按指针或对象引用传递。按引用调用有性能优势,但安全性较低,因为被调函数可以访问原始对象。按常量引用调用则既安全,又有性能优势。

### 软件工程知识

- 6.1 要避免按传值调用传递的开销而且避免调用者的原始数据被修改,可以将长度较长的参数作为常量引用传递。
- 6.2 编写易于理解且维护的程序很重要。改变是常有的事而不是偶尔为之。程序员应该预料到代码会经常修改。可以看出,类程序更易于修改。
- 6.3 类的客户代码使用类对,可以不知道实现类的内部细节。如果类的实现发生改变(例如为了提高性能),只要类的接口保持不变,类的客户源代码就不必变(尽管客户可能需要重新编译),这就使修改系统变得更加容易。
- 6.4 成员函数通常比非面向对象程序中的函数短,因为保存在数据成员中的数据已经由构造函数和保存新数据的成员函数验证。由于数据已经是在对象中,成员函数调用通常不带参数或参数个数至少比在非面向对象语言中的典型函数调用的参数少。所以,调用更简短、函数定义更简化,函数原型也更简化。
- 6.5 客户代码可以访问类的接口,但不能访问类的实现方法。
- 6.6 通过函数原型在类定义中声明成员函数,在类定义之外定义这些函数,可以区分类的接口与实现方法。这样可以实现良好的软件工程。类的客户不能看到类的或成员函数的实现方法,即使实现方法改变,也不需要重新编译。
- 6.7 只有最简单的成员函数和最稳定的或成员函数(即实现方法不会轻易改变)可在类的头文件中定义。
- 6.8 使用面向对象编程方法通常可以减少传递的参数个数,以简化函数调用。面向对象编程好处是对象中封装了数据成员和成员函数,成员函数有权访问数据成员。
- 6.9 本书的中心主题是“复用、复用、再复用”。我们将着重介绍几个提高复用性的技术。我

们将重点放在“建立宝贵的类”和建立宝贵的“软件资产”。

- 6.10 将类声明放在使用类的任意客户代码的头文件中,形成了类的 public 接口(并向客户提供调用类成员函数的函数原型)。将类和成员函数的定义放在源文件中,形成了类的实现方法。
- 6.11 类的客户如果要使用类,不需要访问类的源代码。但是,客户需要连接类的目标码。这样便由独立软件供应商(ISV)提供类库销售或发放许可证。独立软件供应商在其产品中只提供头文件和目标模块,不提供专属信息(例如源代码)。C++ 用户群可以享用更多独立软件供应商生产的类库。
- 6.12 类接口的关键信息应放在头文件中。只用于类内部且类客户不需要的信息应放在不发布的源文件中。这是最低权限原则的另一个例子。
- 6.13 C++ 提倡程序独立于实现方法。独立代码使用的类实现方法发生改变时,无需去改变代码。类接口的任何部分一旦发生改变,独立于实现方法的代码就必须重新编译。
- 6.14 尽量使所有类的数据成员保持 private。提供 public 函数,设置 private 数据成员的值,并获得 private 数据成员的值。这种结构能向类的客户隐藏实现方法,在减少错误的同时,还增强了程序的可修改性。
- 6.15 类设计人员用 private、protected 和 public 成员实施隐藏信息的概念和最低权限原则。
- 6.16 类设计人员不需要提供每个 private 数据成员的 set 和 get 函数,数据成员的设置和 get 函数只有在需要时才提供。如果服务对于客户代码是有用的,就应该在类的 private 接口中提供服务。
- 6.17 将成员函数分成不同的类别:读取和返回 private 数据成员值的函数;设置 private 数据成员值的函数;实现类的服务的函数和进行各种类操作的函数,例如初始化类对象、假设类对象、将类与内部类型和其他类进行相互转换以及处理类对象内存。
- 6.18 面向对象编程的一个现象是,一旦定义了类,生成和操作该类的对象通常只要一个简单的成员函数调用,不需要任何或只需要少量控制结构。相反地,类成员的实现则通常包含控制结构。
- 6.19 如果类的成员函数已经提供类构造函数(或其他成员函数)所需的全部或部分功能,就可以从构造函数(或其他成员函数)中调用成员函数。这不仅简化了代码的维护,还减少了修改代码实现方法时出错的可能性。因此,避免重复代码便成了普遍原则。
- 6.20 如果定义了任何构造函数,并且没有显式的默认构造函数,类就不一定有默认构造函数。
- 6.21 本书稍后将介绍的构造函数和析构函数在C++和面向对象编程中非常重要,这里的简短介绍尚不足以充分说明。
- 6.22 指定 private 数据成员,并通过 public 成员函数控制对这些数据成员的访问,特别是写入访问,可以保证数据的完整性。
- 6.23 通过 set 和 get 函数访问 private 数据不仅能防止数据成员接收无效值,而且还使类的客户忽略数据成员的表达方式。因此,如果数据成员的表达方式因为某种原因而改变(通常为了减少所需存储量或提高性能),只典成员函数提供的接口不变,那么需典改变就只有成员函数,客户代码无须改变,但可能需要重新编译。

## 测试和调试提示

- 6.1 实际上,成员函数调用通常不带参数或其参数比非面向对象语言中的传统函数调用参数要少,其目的是减少传递错误参数、错误参数类型或错误参数个数的可能性。
- 6.2 利用`#ifndef`、`#define`和`#endif`预处理程序指令防止一个程序中多次包含头文件。
- 6.3 将类的数据成员指定为`private`、类的成员函数指定为`public`有助于调试,因为数据操作在类成员函数中或类的友元中局部进行。
- 6.4 每个修改对象`private`数据成员的成员函数(及其友元函数)都应保证该数据的一致性。
- 6.5 指定`private`数据成员并不能自动保证数据的完整性,程序员必须提供有效的检测。但是C++提供了一个框架,有利于程序员用方便的方法设计更好的程序。

## 自测题:

### 6.1 填空题:

- a) 结构定义用关键字\_\_\_\_\_引入。
- b) 类成员通过\_\_\_\_\_操作符和类对象类对象名称的结合使用或通过\_\_\_\_\_操作符和类对象指针的结合使用来进行访问。
- c) 指定为\_\_\_\_\_的类成员只能由类的成员函数和类的友元函数访问。
- d) \_\_\_\_\_是特殊的成员函数,用于初始化类的数据成员。
- e) 类成员的默认访问方式是\_\_\_\_\_。
- f) \_\_\_\_\_函数用于为类的`private`数据成员赋值。
- g) \_\_\_\_\_可用于将一个类对象赋给相同类的另一个类对象。
- h) 类的成员函数通常指定为\_\_\_\_\_,类的数据成员通常指定为\_\_\_\_\_。
- i) \_\_\_\_\_函数用于读取类的`private`数据值。
- j) 类的`public`成员函数集被称为类的\_\_\_\_\_。
- k) 类的实现方法对客户隐藏,也称为\_\_\_\_\_。
- l) 关键字\_\_\_\_\_和\_\_\_\_\_可用于开始类定义。
- m) 指定为\_\_\_\_\_的类成员可以在类对象所在范围中任何位置访问。

### 6.2 指出下列各题中的错误,并说明如何改正。

- a) 假设在`Time`类中声明下列原型:

```
void Time(int);
```

- b) 下面是`Time`类的部分定义:

```
class Time {
public:
    //function prototypes
private:
    int hour = 0;
    int minute = 0;
    int second = 0;
};
```

c) 假设在类 Employee 中声明下列原型:

```
int Employee(const char *, const char *);
```

自测题答案:

- 6.1 a) struct    b) 圆点(. )、箭头(->)    c) private    d) 构造函数    e) private  
f) set    g) 默认按位成员复制(用赋值操作符进行)    h) public, private    i) set  
j) 接口    k) 封装    l) class, struct    m) public

6.2 a) 错误:析构函数不能取得参数或返回数据值。

改正:删除声明中的返回类型 void 和参数 int。

b) 错误:成员不能在类定义中初始化。

改正:从类定义中删除初始化随后在构造函数中初始化数据成员。

c) 错误:构造函数不允许返回数据值。

改正:从声明中删除返回类型 int。

## 练习题

6.3 作用域分辨符有何用途?

6.4 比较C++ 中的 struct 和 class 的概念。

6.5 提供一个构造函数,用 time() 函数中的当前时间初始化 Time 类对象,time() 在 C 标准库的头文件 time.h 中声明。

6.6 生成 complex(复数)类,进行复数运算。编写一个驱动程序,测试这个类。

复数形式

```
realPart + imaginaryPart * i
```

其中 i 为

$$\sqrt{-1}$$

用 double 变量表示类的 private 数据。提供一个构造函数,它可以使这个类的对象在声明时初始化。不提供初始化值时,构造函数就包含默认值。对下列情况提供 public 成员函数:

a) 两个 Complex 值相加:实数部分相加,虚数部分相加。

b) 两个 Complex 值相减:实数部分相减,虚数部分相减。

c) 打印形式如(a,b)的 Complex 值,其中 a 为实数部分,b 为虚数部分。

6.7 生成一个 Rational(有理数)类,执行带分数的运算,编写一个驱动程序,测试这个类。

用整数变量表示类的 private 数据(分子和分母)。提供一个构造函数,当声明类时,使这个类的对象初始化。如果不提供初始化值,构造函数应包含默认值并将分数以简化形式存放。例如分数

$$\frac{2}{4}$$

应在对象中存放成分子 1 和分母 2 的形式。对下列情况提供 public 成员函数:

a) 两个 Rational 值相加,结果以简化形式保存。

b) 两个 Rational 值相加,结果以简化形式保存。

- c) 两个 Rational 值相乘,结果以简化形式保存。
  - d) 两个 Rational 值相除,结果以简化形式保存。
  - e) 按 a/b 形式打印 Rational 值,其中 a 为分子,b 为分母。
  - f) 按浮点数形式打印 Rational 值。
- 6.8 修改图 6.10 中的类 Time,用一个 tick 成员函数将 Time 对象中存放的时间递增 1 秒。Time 对象应该总是保持一致状态。编写一个驱动程序,在循环中测试 tick 成员函数,按标准格式打印时间,从而演示 tick 成员函数的工作情况。要保证测试下列情况:
- a) 递增到下一分钟;
  - b) 递增到下一小时;
  - c) 递增到下一天(11:59:59PM 到 12:00:00AM)。
- 6.9 修改图 6.12 的类 Date,对数据成员 month,day 和 year 的初始化值进行错误检查。提供成员函数 nextDay 将日期递增 1 天。Date 对象应该总是保持一致状态。编写一个驱动程序,在循环中测试 nextDay 成员函数,按标准格式打印日期,演示 nextDay 成员函数的工作情况。要保证测试下列情况:
- a) 递增到下一个月;
  - b) 递增到下一年。
- 6.10 将练习题 6.8 修改的 Time 类和练习题 6.9 修改的 Date 类合并为 DateAndTime 类(第 9 章将介绍继承,继承可以快速完成这项任务而不必修改现有定义)。修改函数 tick,在时间递增到下一天时调用函数 nextDay。修改函数 printStandard 和 printMilitary,同时输出时间和日期。编写一个驱动程序,测试新类 DateAndTime。特别要测试时间递增到下一天时的情况。
- 6.11 修改图 6.10 中的 set 函数,Time 类对象的数据成员设为无效值时返回相应的错误值。
- 6.12 生成一个 Rectangle 类,这个类的 length 和 width 属性默认为 1,其成员函数计算长方形的 perimeter(周长)和 area(面积)。为该类的 length 和 width 设置 set 和 get 函数。set 函数应验证 length 和 width 在 0.0 到 20.0 之间的浮点数。
- 6.13 建立比练习题 6.12 更复杂的 Rectangle 类。这个类只保存长方形 4 个角的直角坐标值。构造函数调用一个 set 函数,接受 4 组坐标并验证它们均在第一象限中,x,y 坐标均不大于 20.0,该函数还验证提供的坐标确实构成长方形。成员函数计算 length,width,perimeter 和 area。长度为两个值中的较大值。用一个判断函数 square 确定是否为正方形。
- 6.14 修改练习题 6.13 中的 Rectangle 类,用函数 draw 在矩形所在第一象限的 25×25 封闭框中显示这个长方形。用函数 setFillCharacter 指定绘制长方形外部的字符。用函数 setPerimeterCharacter 指定绘制长方形 4 个角的字符。还可以加上函数对长方形进行比例缩放、旋转和在第一象限指定范围中移动。
- 6.15 建立 HugeInteger 类,用 40 个元素的数字数组存放最多 40 位的整数组。提供成员函数 inputHugeInteger,outputHugeInteger,addHugeIntegers 和 subtractHugeIntegers。要比较 HugeInteger 对象,提供函数 isEqualTo,isNotEqualTo,isGreaterThan,isLessThan,IsGreaterThanOrEqualTo 和 isLessThanOrEqualTo,两个大数关系成立时这些判断数将返回 true,

关系不成立时,则返回 `false`。该类还提供判断函数 `isZero`,还可以提供成员函数 `multiplyHugeIntegers`, `divideHugeIntegers` 和 `modulusHugeIntegers`。

- 6.16 建立一个类 `TicTacToe`,编写一个三连棋游戏程序。该类包含一个  $3 \times 3$  整型数组,将其作为 `private` 数据,构造函数将空棋盘全部初始化为 0。允许两个人玩游戏。第一个人走棋时,将 1 放在指定格中;第二个人走棋时,将 2 放在指定格中,每次只准移动到空白格。走棋之后,确定是已分胜负(即 3 个 1 连成一条线或 3 个 2 连成一条线),还是出现了平局。还可以将程序修改成人机对战游戏,允许游戏者决定他是先走还是后走。如果有兴趣,还可以开发一个程序,在  $4 \times 4 \times 4$  棋盘上玩 3 维三连棋游戏(注意:这个项目有相当大的难度,开发时间可能长达几个星期)。

## 第7章 类和数据抽象(二)

### 学习目标

- 动态生成与删除对象
- 指定常量对象与常量成员函数
- 了解友元函数和友元类的用途
- 了解如何使用 static 数据成员和成员函数
- 了解容器类的概念
- 了解遍历容器类元素的迭代类概念
- 了解 this 指针的用法

### 7.1 简介

本章继续介绍类和数据抽象。这里将涉及一些较高级的主题,并为第8章介绍类与操作符重载做铺垫。第6章到第8章鼓励程序员使用对象,也就是我们所说的基于对象编程(OBP)。第9章和第10章将介绍继承与多态性,这是真正的面向对象编程(OBP)的技术。在本章及随后的几章中,将使用第5章介绍的C类型的字符串,这样的安排有助于大家掌握C语言指针的复杂性,便于处理近20年来积累的大量C遗留代码。第19章将介绍新的字符串类型,也就是将字符串作为完全成熟的类对象。如此一来,你即可熟悉C++中生成和操作字符串的两种最主要的方法。

### 7.2 常量对象和常量成员函数

我们已经强调过,最低权限原则是良好软件工程的最基本原则之一。下面将介绍如何将这个原则应用于对象。

有些对象需要修改,有些则不需要。程序员用关键字 const 指定不能修改的对象(即常量对象),试图对这类对象做任何修改都会产生语法错误。例如

```
const Time noon(12,0,0);
```

声明 Time 类的常量对象 noon,并将其初始化为中午 12 时。

**软件工程知识 7.1** 将对象声明为 const 有利于实现最低权限原则,试图修改对象会产生编译时错误而非运行时错误。

**软件工程知识 7.2** 使用 const 对于正确的类设计、程序设计和编码非常关键。

**性能提示 7.1** 为变量和对象声明 const 不仅是有效的软件工程原则,还能提高性能,因为如今复杂的优化编译器能对常量进行某些无法对变量进行的优化。



C++ 编译器不允许任何非 const 成员函数调用 const 对象即使获得成员函数并不修改对象。声明为 const 的成员函数不能修改对象,因为编译器不允许这样做。

在函数参数表和函数定义的左花括号之间插入关键字 const,可在原型和定义中将函数指定为 const。例如,下列 A 类的成员函数

```
int A::getValue() const{return privateDataMember};
```

只返回一个对象的数据成员值,并声明为 const。

**常见编程错误 7.1** 把修改对象数据成员的成员函数定义为 const 是语法错误。

**常见编程错误 7.2** 将调用同一类实例的非常量成员函数定义为 const 是语法错误。

**常见编程错误 7.3** 对常量对象调用非常量成员函数是语法错误。

**软件工程知识 7.3** 常量成员函数可以用非常量版本重载。编译器根据对象是否为常量对象自动选择所用的重载版本。

这里产生了一个有关构造函数和析构函数的有趣问题,两者经常需要修改对象。常量对象的构造函数和析构函数不允许有 const 声明。构造函数必须被允许修改对象,以便正确初始化对象。析构函数必须能在对象删除之前进行清理工作。

**常见编程错误 7.4** 将构造函数和析构函数声明为 const 是语法错误。

图 7.1 中的程序实例化两个 Time 对象:一个是非常量对象,一个是常量对象。程序试图用非常量成员函数 setHour(第 102 行)和 printStandard(第 108 行)修改常量对象 noon。程序还演示了 3 个成员函数调用对象的组合:非常量成员函数调用非常量对象(第 100 行),常量成员函数调用非常量成员对象(第 104 行)和常量成员函数调用常量对象(第 106 行和第 108 行)。输出窗口中显示了非常量成员函数调用常量对象时产生的编译错误。

```
1 //Fig. 7.1: time5.h
2 //Declaration of the class Time.
3 //Member functions defined in time5.cpp
4 #ifndef TIMES_H
5 #define TIMES_H
6
7 class Time {
8 public:
9     Time( int = 0, int = 0, int = 0 );    //default constructor
10
11     //set functions
12     void setTime( int, int, int );    //set time
13     void setHour( int );              //set hour
14     void setMinute( int );            //set minute
15     void setSecond( int );            //set second
16
17     //get functions (normally declared const)
18     int getHour() const;               //return hour
19     int getMinute() const;             //return minute
20     int getSecond() const;             //return second
21
```

```

22 //print functions (normally declared const)
23 void printMilitary() const; //print military time
24 void printStandard();      //print standard time
25 private;
26 int hour;                  //0 - 23
27 int minute;                //0 - 59
28 int second;                //0 - 59
29 };
30
31 #endif

```

图 7.1 Time 类的常量对象和常量成员函数用法示例——time5.h

```

32 //Fig. 7.1: time5.cpp
33 //Member function definitions for Time class.
34 #include <iostream>
35
36 using std::cout;
37
38 #include "time5.h"
39
40 //Constructor function to initialize private data.
41 //Default values are 0 (see class definition).
42 Time::Time( int hr, int min, int sec )
43 | setTime( hr, min, sec ); |
44
45 //set the values of hour, minute, and second.
46 void Time::setTime( int h, int m, int s )
47 {
48     setHour( h );
49     setMinute( m );
50     setSecond( s );
51 }
52
53 //set the hour value
54 void Time::setHour( int h )
55 | hour = ( h >= 0 && h < 24 ) ? h : 0; |
56
57 //set the minute value
58 void Time::setMinute( int m )
59 | minute = ( m >= 0 && m < 60 ) ? m : 0; |
60
61 //set the second value
62 void Time::setSecond( int s )
63 | second = ( s >= 0 && s < 60 ) ? s : 0; |
64
65 //get the hour value
66 int Time::getHour() const | return hour; |
67
68 //get the minute value
69 int Time::getMinute() const | return minute; |
70

```

```

71 //get the second value
72 int Time::getSecond() const { return second; }
73
74 //Display military format time: HH:MM
75 void Time::printMilitary() const
76 {
77     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
78         << ( minute < 10 ? "0" : "" ) << minute;
79 }
80
81 //Display standard format time: HH:MM:SS AM (or PM)
82 void Time::printStandard() //should be const
83 {
84     cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
85         << ( minute < 10 ? "0" : "" ) << minute << ":"
86         << ( second < 10 ? "0" : "" ) << second
87         << ( hour < 12 ? " AM" : " PM" );
88 }

```

图 7.1 Time 类的常量对象和常量成员函数用法示例——time5.cpp

```

89 //Fig. 7.1: fig07_01.cpp
90 //Attempting to access a const object with
91 //non-const member functions.
92 #include "time5.h"
93
94 int main()
95 {
96     Time wakeUp( 6, 45, 0 );    //non-constant object
97     const Time noon( 12, 0, 0 ); //constant object
98
99                                     //MEMBER FUNCTION   OBJECT
100     wakeUp.setHour( 18 );    //non-const           non-const
101
102     noon.setHour( 12 );      //non-const           const
103
104     wakeUp.getHour();        //const               non-const
105
106     noon.getMinute();        //const               const
107     noon.printMilitary();    //const               const
108     noon.printStandard();    //non-const           const
109     return 0;
110 }

```

Borland C++ 命令行编译器输出的警告消息:

Fig07\_01.cpp:

```

Warning W8037 Fig07_01.cpp 14;Non-const function Time::setHour(int)
    called for const object in function main()
Warning W8037 Fig07_01.cpp 20;Non-const function Time::printStandard()
    called for const object in function main()
Turbo Incremental Link 5.00 Copyright (c) 1997,2000 Borland

```

Microsoft Visual C++ 编译器输出的错误消息:

```
Compiling...
Fig07_01.cpp
d:\fig07_01.cpp(14):error C2662: 'setHour':cannot convert 'this'
pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
d:\fig07_01.cpp(20):error C2662: 'printStandard':cannot convert
'this' pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
Time5.cpp
Error executing cl.exe.

test.exe -2 error(s),0 warning(s)
```

图 7.1 Time 类的常量对象和常量成员函数用法示例——fig07\_01.cpp

**良好编程习惯 7.1** 将所有无须修改当前对象的成员函数声明为 `const`, 以便在需要时调用常量对象。

注意, 尽管构造函数和析构函数应为非常量成员函数, 但仍可以对常量对象调用构造函数。第 42 行和第 43 行 `Time` 构造函数的定义

```
Time::Time(int hr,int min,int sec)
{ setTime(hr,min,sec); }
```

显示 `Time` 构造函数调用另一个非常量成员函数 `setTime` 对 `Time` 对象进行初始化。允许在常量对象的构造函数调用中调用非常量成员函数。在构造函数完成对象初始化之后, 调用对象的析构函数之间实施对象的常量函数。

**软件工程知识 7.4** 常量对象不能用赋值语句修改, 所以必须初始化。当类的数据成员声明为 `const` 时, 要用成员初始化值向构造函数提供类对象数据成员的初始值。

另外注意第 108 行

```
noon.printStandard(); //non-const const
```

尽管 `Time` 类的成员函数 `printStandard` 没有修改所调用的对象, 但仍然产生了编译错误。没有修改对象并不能充分代表 `const` 方法。 `const` 方法必须显式声明。

图 7.2 演示了用成员初始化值初始化 `Increment` 类的常量数据成员 `increment`。

```
1 //Fig. 7.2: fig07_02.cpp
2 //Using a member initializer to initialize a
3 //constant of a built-in data type.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 class Increment {
10 public:
11     Increment( int c = 0, int i = 1 );
12     void addIncrement() { count += increment; }
```

```

13 void print() const;
14
15 private;
16 int count;
17 const int increment;          //const data member
18 };
19
20 //Constructor for class Increment
21 Increment::Increment( int c, int i )
22     : increment( i ) //initializer for const member
23     { count = c; }
24
25 //Print the data
26 void Increment::print() const
27 {
28     cout << "count = " << count
29         << ", increment = " << increment << endl;
30 }
31
32 int main()
33 {
34     Increment value( 10, 5 );
35
36     cout << "Before incrementing: ";
37     value.print();
38
39     for ( int j = 0; j < 3; j ++ ) {
40         value.addIncrement();
41         cout << "After increment " << j + 1 << ": ";
42         value.print();
43     }
44
45     return 0;
46 }

```

输出结果:

```

Before incrementing:count = 10,increment = 5
After increment 1: count = 15,increment = 5
After increment 2: count = 20,increment = 5
After increment 3: count = 25,increment = 5

```

图 7.2 用成员初始化值初始化内部数据类型的常量

### 修改后的 Increment 构造函数

```

Increment::Increment(int c,int i)
    : increment(i)
    {count = c;}

```

中,符号:increment(i)将 increment 初始化为数值 i。如果需要多个成员初始化值,只需要将其放在冒号后以逗号分隔的表中。所有数据成员都可以用成员初始化值的语法进行初始化,但常量和引用必须用这种方式初始化。本章稍后会介绍,成员对象也必须用这种方式初

始化。派生类的基类部分也要用这种方式初始化。

**测试和调试提示 7.1** 如果成员函数不修改对象,那就始终将其声明为 `const`,以尽量避免错误。

图 7.3 表明了用赋值语句而非成员初始化值初始化 `increment` 时,两个最常用的 C++ 编译器输出的错误消息。

```

1  //Fig. 7.3: fig07_03.cpp
2  //Attempting to initialize a constant of
3  //a built-in data type with an assignment.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  class Increment {
10 public:
11     Increment( int c = 0, int i = 1 );
12     void addIncrement() { count += increment; }
13     void print() const;
14 private:
15     int count;
16     const int increment;
17 };
18
19 //Constructor for class Increment
20 Increment::Increment( int c, int i )
21 {     //Constant member 'increment' is not initialized
22     count = c;
23     increment = i; //ERROR: Cannot modify a const object
24 }
25
26 //Print the data
27 void Increment::print() const
28 {
29     cout << "count = " << count
30         << ", increment = " << increment << endl;
31 }
32
33 int main()
34 {
35     Increment value( 10, 5 );
36
37     cout << "Before incrementing: ";
38     value.print();
39
40     for ( int j = 0; j < 3; j ++ ) {
41         value.addIncrement();
42         cout << "After increment " << j << ": ";
43         value.print();

```

```

44 |
45
46 return 0;
47 |

```

Borland C++ 命令行编译器输出的警告和错误消息:

```

Fig07_03.cpp:
Warning W8038 Fig07_03.cpp 21:Constant member 'Increment::increment'
    is not initialized in function Increment::Increment(int,int)
Error E2024 Fig07_03.cpp 23:Cannot modify a const object in function
    Increment::Increment(int,int)
Warning W8057 Fig07_03.cpp 24:Paramter 'i' is never used in function
    Increment::Increment(int,int)
* * * 1 error in Compile * * *

```

Microsoft Visual C++ 编译器输出的错误消息:

```

Compiling...
Fig07_03.cpp
D:\Fig07_03.cpp(21):error C2758: 'increment';must be initialized in
constructor base/member initializer list
D:\Fig07_03.cpp(16):see diclaration of 'increment'
D:\Fig07_03.cpp(23):error C2166:1-value specifies const object
Error executing cl.exe.

test.exe -2 error(s),0 warning(s)

```

图 7.3 用赋值语句初始化内部数据类型的常量时产生的编译错误

**常见编程错误 7.5** 不为常量数据成员提供成员初始化值是语法错误。

**软件工程知识 7.5** 常量类成员(常量对象和常量变量)要用成员初始化值来初始化,不能用赋值语句。

注意,第 27 行的 print 函数声明为常量,这是因为不会再有常量类型的 Increment 对象。

**软件工程知识 7.6** 如果成员函数不修改对象,最好将所有的类成员函数声明为常量。如果不需要生成该类的常量类型对象,就没有必要这样做。将这种成员函数声明为常量有一个好处,如果无意中修改了这个成员函数中的对象,则编译器会产生语法错误消息。

**测试和调试提示 7.2** 类似 C++ 的语言是不断变化的,新的关键字会不断出现。不要用“object”之类的标识符。尽管“object”目前还不是 C++ 中的关键字,但将来可能会变成关键字,新的编译器可能不会接受现有的代码。

C++ 提供了 mutable 关键字,用于影响程序中常量对象的处理方式。第 21 章将介绍关键字 mutable。

## 7.3 合成:对象作为类成员

AlarmClock 类的对象需要知道什么时候响铃,既然如此,何不将一个 Time 对象作为

AlarmClock 对象的成员呢? 这种功能称为合成。类可以将其他类的对象作为自己的成员。

**软件工程知识 7.7** 合成是软件重用的最常见形式之一, 即一个类可以将其他类的对象作为自己的成员。

生成对象时, 会自动调用其构造函数, 因此需要指定如何将参数传递给成员对象的构造函数。成员对象按声明的顺序(而不是构造函数的成员初始化值列表中的顺序)建立, 而且发生在建立包含他们的类对象(也称为宿主对象)之前。

图 7.4 中的程序用 Employee 类和 Date 类演示一个类作为其他类对象的成员。Employee 类包含 private 数据成员 firstName, lastName, birthDate 和 hireDate。成员 birthDate 和 hireDate 是 Date 类的 const 类型对象, 该 Date 类包含 private 数据成员 month, day 和 year。程序实例化一个 Employee 对象, 初始化并显示其数据成员。注意 Employee 构造函数定义中函数头文件的语法

```
Employee::Employee(char * fname, char * lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear)
: birthDate(bmonth, bday, byear),
  hireDate(hmonth, hday, hyear)
```

该构造函数有 8 个参数 (fname, lname, bmonth, bday, byear, hmonth, hday 和 hyear)。头文件中, 用冒号(:)分隔成员初始化值与参数表。成员初始化值指定 Employee 的参数传递给成员 Date 对象的构造函数, 参数 bmonth, bday 和 byear 传递给对象 birthDate 的构造函数。参数 hmonth, hday 和 hyear 传递给对象 hireDate 的构造函数, 用逗号分隔多个成员的初始化值。

```
1 //Fig. 7.4: date1.h
2 //Declaration of the Date class.
3 //Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 / class Date {
8 public:
9     Date( int = 1, int = 1, int = 1900 ); //default constructor
10    void print() const; //print date in month/day/year format
11    ~Date(); //provided to confirm destruction order
12 private:
13    int month; //1-12
14    int day; //1-31 based on month
15    int year; //any year
16
17    //utility function to test proper day for month and year
18    int checkDay( int );
19 };
20
21 #endif
```

图 7.4 使用成员对象的初始化值——date1.h



```
22 //Fig. 7.4: datel.cpp
23 //Member function definitions for Date class.
24 #include <iostream>
25
26 using std::cout;
27 using std::endl;
28
29 #include "datel.h"
30
31 //Constructor: Confirm proper value for month;
32 //call utility function checkDay to confirm proper
33 //value for day.
34 Date::Date( int mn, int dy, int yr )
35 {
36     if ( mn > 0 && mn <= 12 )           //validate the month
37         month = mn;
38     else {
39         month = 1;
40         cout << "Month " << mn << " invalid. set to month 1. \n";
41     }
42
43     year = yr;                          //should validate yr
44     day = checkDay( day );              //validate the day
45
46     cout << "Date object constructor for date ";
47     print(); //interesting: a print with no arguments
48     cout << endl;
49 }
50
51 //Print Date object in form month/day/year
52 void Date::print() const
53 { cout << month << '/' << day << '/' << year; }
54
55 //Destructor: provided to confirm destruction order
56 Date::~Date()
57 {
58     cout << "Date object destructor for date ";
59     print();
60     cout << endl;
61 }
62
63 //Utility function to confirm proper day value
64 //based on month and year.
65 //Is the year 2000 a leap year?
66 int Date::checkDay( int testDay )
67 {
68     static const int daysPerMonth[ 13 ] =
69         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
70
71     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
72         return testDay;
```

```

73
74  if ( month == 2 && //February: Check for leap year
75      testDay == 29 &&
76      ( year % 400 == 0 ||
77        ( year % 4 == 0 && year % 100 != 0 ) ) )
78      return testDay;
79
80  cout << "Day " << testDay << " invalid. set to day 1. \n";
81
82  return 1; //leave object in consistent state if bad value
83 }

```

图 7.4 使用成员对象的初始化值——date1.cpp

```

84 //Fig. 7.4; empty1.h
85 //Declaration of the Employee class.
86 //Member functions defined in empty1.cpp
87 #ifndef EMPLOY1_H
88 #define EMPLOY1_H
89
90 #include "date1.h"
91
92 class Employee {
93 public:
94     Employee( char *, char *, int, int, int, int, int, int );
95     void print() const;
96     ~Employee(); //provided to confirm destruction order
97 private:
98     char firstName[ 25 ];
99     char lastName[ 25 ];
100     const Date birthDate;
101     const Date hireDate;
102 };
103
104 #endif

```

图 7.4 使用成员对象的初始化值——empty1.h

```

105 //Fig. 7.4; empty1.cpp
106 //Member function definitions for Employee class.
107 #include <iostream>
108
109 using std::cout;
110 using std::endl;
111
112 #include <cstring>
113 #include "empty1.h"
114 #include "date1.h"
115
116 Employee::Employee(char * fname, char * lname,
117                   int bmonth, int bday, int byear,
118                   int hmonth, int hday, int hyear )
119     : birthDate( bmonth, bday, byear ),

```

```

120     hireDate( hmonth, hday, hyear )
121 }
122     //copy fname into firstName and be sure that it fits
123     int length = strlen( fname );
124     length = ( length < 25 ? length : 24 );
125     strncpy( firstName, fname, length );
126     firstName[ length ] = '\0';
127
128     //copy lname into lastName and be sure that it fits
129     length = strlen( lname );
130     length = ( length < 25 ? length : 24 );
131     strncpy( lastName, lname, length );
132     lastName[ length ] = '\0';
133
134     cout << "Employee object constructor: "
135           << firstName << " " << lastName << endl;
136 }
137
138 void Employee::print() const
139 {
140     cout << lastName << ", " << firstName << "\nHired: ";
141     hireDate.print();
142     cout << " Birth date: ";
143     birthDate.print();
144     cout << endl;
145 }
146
147 //Destructor; provided to confirm destruction order
148 Employee::~Employee()
149 {
150     cout << "Employee object destructor: "
151           << lastName << ", " << firstName << endl;
152 }

```

图 7.4 使用成员对象的初始化值——empty1.cpp

```

153 //Fig. 7.4: fig07_04.cpp
154 //Demonstrating composition; an object with member objects.
155 #include <iostream>
156
157 using std::cout;
158 using std::endl;
159
160 #include "empty1.h"
161
162 int main()
163 {
164     Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
165
166     cout << '\n';
167     e.print();
168 }

```

```

169     cout << " \nTest Date constructor with invalid values:\n";
170     Date d( 14, 35, 1994 ); //invalid Date values
171     cout << endl;
172     return 0;
173 }

```

输出结果:

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor:Bob Jones

```

```

Jones,Bob
Hired:3/12/1988 Birth date:7/24/1949

```

```

Test Date constructor with invalid values:
Month 14 invalid.set to month 1.
Day 35 invalid.set to day 1.
Date object constructor for date 1/1/1994

```

```

Date object destructor for date 1/1/1994
Employee object destructor:Jones,Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1994

```

图 7.4 使用成员对象的初始化值——fig07\_1.cpp

记住,常量成员和引用也在成员初始化值列表中初始化(第9章将介绍的派生类基类也如此)。Date类和Employee类各有一个析构函数,分别在删除Date对象和Employee对象时打印一条信息。这使我们可以通过程序输出确定对象由内向外建立,由外向内删除(即先删除Employee对象,再删除其中包含的Date对象)。

成员对象不需要通过成员初始化值显式初始化。如果没有提供成员初始化值,将隐式调用成员对象的默认构造函数。默认构造函数建立的值(如果有)可以用set函数重定义。但这种方法用于进行复杂的初始化非常耗费时间和精力。

**常见编程错误 7.6** 既不为成员对象提供成员初始化值,又不为成员对象的类提供默认的构造函数会造成语法错误。

**性能提示 7.2** 通过成员初始化值显式初始化成员对象,可以避免两次初始化成员对象的开销:一次是调用成员对象的默认构造函数,另一次是用set函数初始化成员对象。

**软件工程知识 7.8** 如果一个类将其他类的对象作为其成员,即使将该这个成员对象指定为public,也不会破坏该成员对象private成员的封装与隐藏。

注意,第52行调用Date成员函数print。C++中许多类的成员函数都不需要参数,这是因为每个成员函数包含所操作对象的隐式句柄(指针形式)。7.5节将介绍this隐式指针。

为简化编程,这里Employee类的第一个版本中,我们用两个25个字符的数组表示Employee的姓和名。这些数组如果存储小于24个字符的名称就会浪费内存空间(记住每个数组中有一个字符是字符串的空中止符'\0')。同时,超过24个字符的姓名要截尾才能放

入字符数组中。本章稍后将会介绍 `Employee` 类的另一版本,动态生成适合姓和名长度的数组,可以用两个 `string` 对象表示姓名。第 19 章将详细介绍 `string` 标准库类。

## 7.4 友元函数和友元类

类的友元函数在类作用域之外定义,但有权访问类的 `private` 和第 9 章将介绍的 `protected` 成员。函数或整个类都可以声明为另一个类的友元函数或类。

使用友元函数可提高性能。这里将介绍一个友元函数的例子。本书后面将介绍如何通过类对象,用友元函数重载操作符和生成迭代类。迭代类的对象用于连续选择项目或对容器类(见 7.9 节)对象中的项目进行操作。容器类对象能够存放项目。成员函数无法进行某些操作时,可以使用友元函数(详情参见第 8 章)。

要将一个函数声明为类的友元,应在类定义的函数原型前加上关键字 `friend`。要将 `ClassTwo` 类声明为 `ClassOne` 类的友元,应在 `ClassOne` 类的定义中进行如下声明

```
friend class ClassTwo;
```

**软件工程知识 7.9** 尽管类定义中有友元函数的原型,但友元函数仍然不是成员函数。

**软件工程知识 7.10** `private`, `protected` 和 `public` 的成员访问符号与友元关系的声明无关,因此友元关系声明可以放入类定义中的任何位置。

**良好编程习惯 7.2** 将类中所有友元关系的声明放在类的头文件之后,不要在其前面添加任何成员访问说明符。

友元关系是指定的,不是获取的,如果要让类 `B` 成为类 `A` 的友元类,类 `A` 必须显式声明类 `B` 为自己的友元类。此外,友元关系既不对称也不能传递。例如,如果类 `A` 是类 `B` 的友元类,类 `B` 是类 `C` 的友元类,并不能认为类 `B` 是类 `A` 的友元类(不对称),类 `C` 是类 `B` 的友元类或类 `A` 是类 `C` 的友元类(不传递)。

**软件工程知识 7.11** OOP 组织中,有人认为友元关系破坏了信息隐藏并削弱了面向对象设计方法的优势。

图 7.5 演示了声明与使用友元函数 `setX` 来设置 `count` 类的 `private` 数据成员 `x`。注意, `friend` 声明出现在类定义的开始位置(习惯上是这样),甚至放在 `public` 成员函数的声明之前。图 7.6 演示了调用非友元函数 `cannotsetx` 修改 `private` 数据成员 `x` 时编译器产生的信息。图 7.5 和图 7.6 将介绍使用友元函数的“结构”,随后将展示几个友元函数用法示例。

```
1 //Fig. 7.5: fig07_05.cpp
2 //Friends can access private members of a class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 //Modified Count class
9 class Count {
```

```

10 friend void setX( Count &, int ); //friend declaration
11 public:
12     Count() { x = 0; } //constructor
13     void print() const { cout << x << endl; } //output
14 private:
15     int x; //data member
16 };
17
18 //Can modify private data of Count because
19 //setX is declared as a friend function of Count
20 void setX( Count &c, int val )
21 {
22     c.x = val; //legal; setX is a friend of Count
23 }
24
25 int main()
26 {
27     Count counter;
28
29     cout << "counter.x after instantiation: ";
30     counter.print();
31     cout << "counter.x after call to setX friend function: ";
32     setX( counter, 8 ); //set x with a friend
33     counter.print();
34     return 0;
35 }

```

输出结果:

```

counter.x after instantiation:0
counter.x after call to setX friend function:8

```

图 7.5 友元函数可以访问类的 private 成员

注意,函数 setX(第 20 行)是 C 类型的单独函数,而不是 Count 类的成员函数。为此,counter 对象调用 setX 时,我们使用了第 32 行

```
setX( counter, 8 ); //set x with a friend
```

该语句取 counter 作为参数,而不是像语句

```
counter.setX(8);
```

一样用句柄(如对象名)调用函数。如前所述,图 7.5 是友元构造函数的一个实例,通常适合将函数 setX 定义为 Count 类的成员函数。

```

1  /Fig. 7.6: fig07_06.cpp
2  //Non-friend/non-member functions cannot access
3  //private data of a class.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  //Modified Count class

```

```

10 class Count {
11 public:
12     Count() { x = 0; }           //constructor
13     void print() const { cout << x << endl; } //output
14 private:
15     int x; //data member
16 };
17
18 //Function tries to modify private data of Count,
19 //but cannot because it is not a friend of Count.
20 void cannotsetX( Count &c, int val )
21 {
22     c.x = val; //ERROR: 'Count::x' is not accessible
23 }
24
25 int main()
26 {
27     Count counter;
28
29     cannotsetX( counter, 3 ); //cannotsetX is not a friend
30     return 0;
31 }

```

**Borland C++ 命令行编译器输出的错误消息:**

```

Borland C++ 5.5 for Win32 Copyright (c) 1993,2000 Borland
Fig07_06.cpp:
Error E2247 Fig07_06.cpp 22: 'Count::x' is not accessible in
    function cannotsetX(Count &,int)
*** 1 errors in Compile ***

```

**Microsoft Visual C++ 编译器输出的错误消息:**

```

Compiling...
Fig07_06.cpp
D:\books\2000\cpp\http3\examples\Ch07\Fig07_06\Fig07_06.cpp(22):
    error C2248: 'x':cannot access private member declared in
    class 'Count'
        D:\books\2000\cpp\http3\examples\Ch07\Fig07_06\
        Fig07_06.cpp(15):see declaration of 'x'
Error executing cl.exe.

test.exe -1 error(s),0 warning(s)

```

**图 7.6 非友元函数、非成员函数不能访问类的 private 成员**

**软件工程知识 7.12** 由于C++属于混合语言,常在同一个程序中采用两种函数调用且这两种函数调用往往是相反的,类C语言的调用将基本数据或对象传递给函数,C++调用则是将函数(或信息)传递给对象。

重载函数可以指定为类的友元。每个重载函数要作为友元,须在类定义中显式声明为类的友元。

## 7.5 使用 this 指针

每个对象都可以通过 this 指针访问自己的地址。对象的 this 指针不属于对象本身的一部分,即 this 指针不会出现在该对象的 sizeof 操作结果中。但 this 指针会在每次非 static 成员(详情参见 7.7 节)函数调用对象时作为第一个隐式参数传递给对象(通过编译器)。

this 指针隐式引用对象的数据成员函数,当然也可以显式引用。this 指针的类型取决于对象的类型和使用 this 的成员函数是否声明为常量。在类 Employee 的非常量成员函数中, this 指针的类型为 Employee \* const( Employee 对象的常量指针)。在类 Employee 的常量成员函数中, this 指针的类型为 const Employee \* const(为常量 Employee 对象的常量指针)。

下面将介绍显式使用 this 指针的简单示例,本章稍后以及第 8 章将介绍一些较为复杂的 this 用法示例。每个非静态成员函数都能访问被调成员所在对象的 this 指针。

**性能提示 7.3** 为节约内存空间,每个类的每个成员都只有一个副本,该类的每个对象都可调用这个成员函数。另一方面,每个对象又有自己的类数据成员副本。

图 7.7 演示了显式使用 this 指针,令 Test 类的成员函数打印 Test 对象的 private 数据 x。

```

1 //Fig. 7.7: fig07_07.cpp
2 //Using the this pointer to refer to object members.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Test {
9 public:
10     Test( int = 0 );           //default constructor
11     void print() const;
12 private:
13     int x;
14 };
15
16 Test::Test( int a ) { x = a; } //constructor
17
18 void Test::print() const       //( ) around *this required
19 {
20     cout << " x = " << x
21         << "\n this ->x = " << this ->x
22         << "\n (*this).x = " << ( *this ).x << endl;
23 }
24
25 int main()
26 {
27     Test testObject( 12 );
28
29     testObject.print();

```



```

30
31     return 0;
32 }

```

输出结果:

```

        x = 12
    this->x = 12
    (*this).x = 12

```

图 7.7 this 指针用法示例

为进一步说明,图 7.7 中的 print 成员函数首先直接打印 x,然后 print 用两个不同符号通过 this 指针访问 x:this 指针与箭头操作符(→)和复引用 this 指针与圆点操作符(.)。

注意,\*this 和圆点(成员选择)操作符一起使用时,\*this 要用括号括起来。括号是必须的,因为圆点操作符的优先级高于\*操作符。如果不用括号,表达式

```
*this.x
```

就会视为使用了圆括号的表达式

```
*(this.x)
```

进行求值该表达式有语法错误,因为圆点操作符不能与指针一起使用。

**常见编程错误 7.7** 打算同时使用对象指针和成员选择操作符(.)是语法错误,因为成员选择操作符和对象成该对象的引用一起使用。

this 指针的一个有趣用法是防止对象自我赋值。对象包含动态分配内存的指针时自我赋值可能会导致严重错误,详情参见第 8 章。

this 指针的另一个用法是连续使用成员函数调用。图 7.8 演示了返回 Time 对象的引用,使 Time 类的成员调用可以连续使用。成员函数 setTime, setHour, setMinute 和 setSecond 都可以返回 Time & 返回类型的 \*this。

```

1 //Fig. 7.8; time6.h
2 //Cascading member function calls.
3
4 //Declaration of class Time.
5 //Member functions defined in time6.cpp
6 #ifndef TIME6_H
7 #define TIME6_H
8
9 class Time {
10 public:
11     Time( int = 0, int = 0, int = 0 );    //default constructor
12
13     //set functions
14     Time &setTime( int, int, int );        //set hour, minute, second
15     Time &setHour( int );                  //set hour
16     Time &setMinute( int );                //set minute
17     Time &setSecond( int );                //set second
18
19     //get functions (normally declared const)
20     int getHour() const;                   //return hour

```

```

21  int getMinute() const;           //return minute
22  int getSecond() const;          //return second
23
24  //print functions (normally declared const)
25  void printMilitary() const;      //print military time
26  void printStandard() const;     //print standard time
27 private:
28  int hour;                        //0 - 23
29  int minute;                      //0 - 59
30  int second;                     //0 - 59
31 };
32
33 #endif

```

图 7.8 连续使用成员函数调用——time6.h

```

34 //Fig. 7.8: time6.cpp
35 //Member function definitions for Time class.
36 #include <iostream>
37
38 using std::cout;
39
40 #include "time6.h"
41
42 //Constructor function to initialize private data.
43 //Calls member function setTime to set variables.
44 //Default values are 0 (see class definition).
45 Time::Time( int hr, int min, int sec )
46 { setTime( hr, min, sec ); }
47
48 //set the values of hour, minute, and second.
49 Time &Time::setTime( int h, int m, int s )
50 {
51     setHour( h );
52     setMinute( m );
53     setSecond( s );
54     return *this; //enables cascading
55 }
56
57 //set the hour value
58 Time &Time::setHour( int h )
59 {
60     hour = ( h >= 0 && h < 24 ) ? h : 0;
61
62     return *this; //enables cascading
63 }
64
65 //set the minute value
66 Time &Time::setMinute( int m )
67 {
68     minute = ( m >= 0 && m < 60 ) ? m : 0;

```

```

69
70     return *this; //enables cascading
71 }
72
73 //set the second value
74 Time &Time::setSecond( int s )
75 {
76     second = ( s >= 0 && s < 60 ) ? s : 0;
77
78     return *this; //enables cascading
79 }
80
81 //get the hour value
82 int Time::getHour() const { return hour; }
83
84 //get the minute value
85 int Time::getMinute() const { return minute; }
86
87 //get the second value
88 int Time::getSecond() const { return second; }
89
90 //Display military format time; HH;MM
91 void Time::printMilitary() const
92 {
93     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
94           << ( minute < 10 ? "0" : "" ) << minute;
95 }
96
97 //Display standard format time; HH;MM;SS AM (or PM)
98 void Time::printStandard() const
99 {
100     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
101           << ":" << ( minute < 10 ? "0" : "" ) << minute
102           << ":" << ( second < 10 ? "0" : "" ) << second
103           << ( hour < 12 ? " AM" : " PM" );
104 }

```

图 7.8 连续使用成员函数调用——time6.cpp

```

105 //Fig. 7.8; fig07_08.cpp
106 //Cascading member function calls together
107 //with the this pointer
108 #include <iostream>
109
110 using std::cout;
111 using std::endl;
112
113 #include "time6.h"
114
115 int main()
116 {
117     Time t;
118

```

```

119 t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
120 cout << "Military time: ";
121 t.printMilitary();
122 cout << "\nStandard time: ";
123 t.printStandard();
124
125 cout << "\n\nNew standard time: ";
126 t.setTime( 20, 20, 20 ).printStandard();
127 cout << endl;
128
129 return 0;
130 }

```

输出结果:

Military time: 18:30

Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

图 7.8 连续使用成员函数调用——fig07\_08.cpp

为什么可以将 \*this 作为引用返回呢? 圆点操作符(.)的结合性是从左向右,所以下列表达式

```
t.setHour(18).setMinute(30).setSecond(22);
```

首先求值 t.setHour(18),然后返回对象 t 的引用,将其作为这个函数调用的值,其余表达式解释如下

```
t.setMinute(30).set(22);
```

执行 setMinute(30)调用并返回 t 的等价值,其余表达式解释为

```
t.setSecond(32);
```

注意下列调用

```
t.set(20,20,20).printStandard();
```

也体现了连续使用的特性。这些调用在表达式中必须以这个顺序出现。因为类中定义的 printStandard 没有返回 t 的引用,所以把上述语句中的 printStandard 放在 setTime 之前是语法错误。

## 7.6 用 new 和 delete 实现动态内存分配

和 C 语言的 malloc 与 free 函数调用相比,new 和 delete 操作符提供的动态分配内存方法更好(对任何内部或用户自定义类型而言)。以代码

```
TypeName *typeNamePtr; .
```

为例,在 ANSI C 语言中,要动态生成 TypeName 类型的对象,须用语句

```
typeNamePtr = malloc(sizeof(TypeName));
```

该语句要求调用 malloc 函数和显式使用 sizeof 操作符。在 ANSI C 之前的 C 语言版本中,还要对 malloc 返回的指针进行类型转换(利用 TypeName \*)。malloc 函数没有提供任何初始

化已分配内存块的方法。在C++中,只须简单写出语句

```
typeNamePtr = new TypeName;
```

new 操作符自动生成长度正确的对象并调用对象的构造函数和返回正确类型的指针。在ANSI/ISO C++ 标准之前的C++ 版本中,如果 new 无法找到内存空间,它将返回 0 指针(注意,第 13 章将介绍 ANSI/ISO C++ 标准中如何处理 new 失败。特别要介绍 new 如何“抛出”异常,如何“捕捉”并处理异常)。C++ 中要删除对象并释放其空间时,必须用 delete 操作符,如下所示

```
delete typeNamePtr;
```

C++ 允许对新生成的对象提供初始化值,如语句

```
Double *thingPtr = new double(3.14159);
```

它将新生成的 double 对象初始化为 3.14159。

可以生成 10 个元素的整型数组并将其赋给 arrayPtr,如下所示

```
int *arrayPtr = new int[10];
```

该数组可以用语句

```
delete[] arrayPtr;
```

删除。如后文所述,用 new 和 delete 代替 malloc 和 free 还有别的好处。这样的好处尤其体现于 new 调用类的构造函数,delete 调用类的析构函数时。

**常见编程错误 7.8** 将 new 和 delete 类型的动态分配内存方法与 malloc 和 free 类型的动态分配内存方法混用是逻辑错误:malloc 分配的空间无法用 delete 释放,new 分配的空间无法用 free 删除。

**常见编程错误 7.9** 删除数组时,用 delete 代替 delete[] 将导致运行时的逻辑错误。为避免这一错误,数组生成的内存空间要用 delete[] 操作符删除,各个元素生成的内存空间则用 delete 操作符删除。

**良好编程习惯 7.3** 因为C++ 语言包含 C 语言,所以C++ 程序可以同时包含用 malloc 生成和用 free 删除的存储空间以及用 new 生成和用 delete 删除的对象。但最好尽量使用 new 和 delete。

## 7.7 静态类成员

类的每个对象都有其所有数据成员的副本。某些情况下只有一个变量副本供类的所有对象共享。静态(static)变量就是为此以及其他用途而设计的。静态类变量表示整个类范围中(所有类对象而非指定的类对象)共享的信息。静态成员的声明用关键字 static 开始。

下面以一个视频游戏为例,说明静态类共享数据的用途。假设视频游戏中有 Martian 和其他太空人。每个 Martian 都很勇敢,当 Martian 知道至少有 5 个 Martian 存在时,就要攻击其他太空人。如果 Martian 的人数少于 5 个,Martian 就会胆小而不敢攻击。因此每个 Martian 需要知道 martianCount。我们在类 Martian 中提供一个 martianCount 数据成员。这样,每个 Martian 就会有该数据成员的副本,每次生成新 Martian 时,就要更新每个 Martian 对象中

的 `martianCount` 数据成员,这样既浪费空间又浪费时间。为此,我们将 `martianCount` 声明为 `static`,以使 `martianCount` 成为类共享数据。每个 `Martian` 都可以访问 `martianCount`,就像是自己的数据成员一样,但C++ 只需维护 `martianCount` 的一个静态副本,这样可以节省空间。让 `Martian` 构造函数递增静态 `martianCount` 还能节省时间,因为只有一个副本,无须再对每个 `Martian` 对象递增 `martianCount`。

**性能提示 7.4** 一个数据副本够用时,用静态数据成员则可以节省存储空间。

尽管静态数据成员看上去像是全局变量,但 `static` 数据成员有类作用域,静态成员可以是 `public`,`private` 或 `protected`。静态数据成员在文件范围内必须进行一次初始化。类的 `public` 静态类成员可以通过类的任何对象访问,也可以用二元作用域分辨符通过类名进行访问。类的 `private` 和 `protected` 静态成员必须通过类的 `public` 成员函数或类的友元访问。即使类没有对象,但仍然有静态成员。类没有对象时,要想访问 `public` 静态类成员,只需在成员数据名前面加上类名和二元作用域分辨符(`::`)。要在类没有对象时访问 `private` 或 `protected` 静态类成员,则需要提供一个 `public` 静态成员函数,并在调用函数时在函数名前面加上类名和二元作用域分辨符。

图 7.9 中的程序演示了 `private` 静态数据成员和 `public` 静态成员函数的用法。数据成员 `count` 在文件范围内用语句

```
int Employee::count = 0;
```

初始化为 0。数据成员 `count` 维护 `Employee` 类实例化的对象个数。有 `Employee` 类的对象时,可通过 `Employee` 对象的任何成员函数引用 `count` 成员。本例中,构造函数和析构函数都引用了 `count`。

```
1 //Fig. 7.9: employ1.h
2 //An employee class
3 #ifndef EMPLOY1_H
4 #define EMPLOY1_H
5
6 class Employee {
7 public:
8     Employee( const char *, const char * ); //constructor
9     ~Employee(); //destructor
10    const char *getFirstName() const; //return first name
11    const char *getLastName() const; //return last name
12
13    //static member function
14    static int getCount(); //return # objects instantiated
15
16 private:
17     char *firstName;
18     char *lastName;
19
20     //static data member
21     static int count; //number of objects instantiated
22 };
23
```

24 #endif

图 7.9 用静态数据成员维护类对象的个数——employ1.h

```

25 //Fig. 7.9: employ1.cpp
26 //Member function definitions for class Employee
27 #include <iostream>
28
29 using std::cout;
30 using std::endl;
31
32 #include <cstring>
33 #include <cassert>
34 #include "employ1.h"
35
36 //Initialize the static data member
37 int Employee::count = 0;
38
39 //Define the static member function that
40 //returns the number of employee objects instantiated.
41 int Employee::getCount() { return count; }
42
43 //Constructor dynamically allocates space for the
44 //first and last name and uses strcpy to copy
45 //the first and last names into the object
46 Employee::Employee( const char *first, const char *last )
47 {
48     firstName = new char[ strlen( first ) + 1 ];
49     assert( firstName != 0 ); //ensure memory allocated
50     strcpy( firstName, first );
51
52     lastName = new char[ strlen( last ) + 1 ];
53     assert( lastName != 0 ); //ensure memory allocated
54     strcpy( lastName, last );
55
56     ++count; //increment static count of employees
57     cout << "Employee constructor for " << firstName
58         << " " << lastName << " called." << endl;
59 }
60
61 //Destructor deallocates dynamically allocated memory
62 Employee::~Employee()
63 {
64     cout << " ~Employee() called for " << firstName
65         << " " << lastName << endl;
66     delete[] firstName; //recapture memory
67     delete[] lastName; //recapture memory
68     --count; //decrement static count of employees
69 }
70
71 //Return first name of employee
72 const char *Employee::getFirstName() const

```

```

73 |
74 //Const before return type prevents client from modifying
75 //private data. Client should copy returned string before
76 //destructor deletes storage to prevent undefined pointer.
77 return firstName;
78 |
79
80 //Return last name of employee
81 const char *Employee::getLastName() const
82 {
83 //Const before return type prevents client from modifying
84 //private data. Client should copy returned string before
85 //destructor deletes storage to prevent undefined pointer.
86 return lastName;
87 }

```

图 7.9 用静态数据成员维护类对象的个数——employ1.cpp

```

88 //Fig. 7.9: fig07_09.cpp
89 //Driver to test the employee class
90 #include <iostream>
91
92 using std::cout;
93 using std::endl;
94
95 #include "employ1.h"
96
97 int main()
98 |
99     cout << "Number of employees before instantiation is "
100         << Employee::getCount() << endl; //use class name
101
102     Employee *e1Ptr = new Employee( "Susan", "Baker" );
103     Employee *e2Ptr = new Employee( "Robert", "Jones" );
104
105     cout << "Number of employees after instantiation is "
106         << e1Ptr ->getCount();
107
108     cout << "\n\nEmployee 1: "
109         << e1Ptr ->getFirstName()
110         << " " << e1Ptr ->getLastName()
111         << "\nEmployee 2: "
112         << e2Ptr ->getFirstName()
113         << " " << e2Ptr ->getLastName() << "\n\n";
114
115     delete e1Ptr; //recapture memory
116     e1Ptr = 0;
117     delete e2Ptr; //recapture memory
118     e2Ptr = 0;
119
120     cout << "Number of employees after deletion is "
121         << Employee::getCount() << endl;

```



```

122
123     return 0;
124 }

```

输出结果:

```

Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

```

```

Employee 1: Susan Baker
Employee 2: Robert Jones

```

```

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0

```

图 7.9 用静态数据成员维护类对象的个数——fig07\_09.cpp

**常见编程错误 7.10** 文件范围内 static 类变量定义中使用关键字 static 是语法错误。

Employee 类没有对象时,仍然可以引用 count 成员,但是只能通过调用静态成员函数 getCount 进行,如下所示

```
Employee::getCount()
```

本例中,函数 getCount 用于确定当前实例化的 Employee 对象个数。注意,程序中没有实例化的对象时,执行的是 Employee::getCount() 函数调用。但如果有实例化的对象,则可以通过第 105 行和第 106 行

```

cout << "Number of employees after instantiation is "
      << e1Ptr ->getCount();

```

中的其中一个对象调用了函数 getCount。注意,调用 e2Ptr -> getCount() 和 Employee::getCount() 会输出同样的结果。

**软件工程知识 7.13** 有些公司的软件工程标准中,明确规定所有静态成员函数只能调用类名句柄,不能调用对象句柄。

如果成员函数不访问非静态类数据成员和成员函数,可以将成员函数声明为静态。与非静态成员函数不同的是,静态成员函数没有 this 指针,因为静态类数据成员和成员函数是独立于类对象而存在的。

**常见编程错误 7.11** 在静态成员函数中引用 this 指针是语法错误。

**常见编程错误 7.12** 将静态成员函数声明为常量是语法错误。

**软件工程知识 7.14** 即使类对象尚未初始化类的静态数据成员和成员函数也可以已经存在并可使用。

第 102 行和第 103 行用 new 操作符动态分配两个 Employee 对象。分配每个 Employee 对象时,调用其构造函数。第 115 行和第 117 行用 delete 释放两个 Employee 对象的内存空间时,调用其析构函数。

**良好编程习惯 7.4** 删除动态分配内存后,将指向该内存的指针设置为指向0,以切断指针与前面已分配内存的连接。

注意,Employee 的构造函数使用了 `assert`(断言)。`assert` 类宏在 `Cassert` 头文件中定义,用以测试条件值。如果表达式的值为 `false`,`assert` 就会发出错误信息,并调用 `abort` 函数(在常用工具程序头文件 `<cstdlib>` 中)中止程序执行。这是个有用的调试工具,可以测试变量是否有正确值。注意,函数 `abort` 不运行任何析构函数即可中止程序执行。

在这个程序中,`assert` 用于确定 `new` 操作符能否满足动态分配内存的请求。例如,在 Employee 构造函数中,语句(也称为断言):

```
assert( firstName != 0);
```

用于测试指针 `firstName`,确定其是否不等于0。如果上述 `assert` 中的条件为 `true`,程序将继续执行,不被中断。如果上述 `assert` 中的条件为 `false`,程序就会打印出一条错误信息,包括行号、测试条件和 `assert` 所在的文件名,然后程序中止。程序员可以从这个代码段中找出错误。第13章将介绍执行时错误的更好处理方法。

`assert` 不一定要在调试完成后删除。程序不再用 `assert` 进行调试时,只须在程序文件开头(通常可以在编译器选项中指定)插入语句

```
#define NDEBUG
```

这时预处理程序会忽略所有断言,无须程序员手工删除各条断言。

注意,函数 `getFirstName` 和 `getLastName` 的实现方法向类的客户代码返回常量字符指针。在这个实现方法中,如果客户要保留姓和名的副本,就应在取得对象的常量字符指针之后负责复制 Employee 对象的动态分配内存。注意,还可以用 `getFirstName` 和 `LastName` 让客户代码向每个函数传递字符数组和数组长度。然后,函数将姓或名复制到客户代码提供的字符数组中。此外,`strint` 类(参见第19章)还可用于将字符串对象的副本返回主调函数。

## 7.8 数据抽象和信息隐藏

类通常对类客户代码隐藏其实现方法的细节,即所谓的信息隐藏。下面将以堆栈数据结构作为信息隐藏的例子。堆栈可以看作一堆盘子。盘子放入堆中时,始终放在顶部(压入堆栈);从堆中取出盘子时,始终从顶部取(称为弹出堆栈)。也就是说,堆栈是先进后出的数据结构,最后放进(插入)堆栈的项目始终最先从堆栈中取出(移除)。

程序员可以生成堆栈类,并向客户隐藏实现细节。堆栈可以方便地用数组或用第15章“数据结构”介绍的链表来实现。堆栈类的客户无须得知堆栈如何实现,只要求数据项目在放入堆栈中时,会按后进先出的方法重新调用。描述与其实现细节无关的类的功能称为数据抽象,C++ 的类定义了所谓的抽象数据类型(ADT)。尽管用户可能知道类的实现细节,但不应编写依赖这些实现细节的代码。这意味着只要其 `public` 接口不变,特定类(如实现堆栈与其“压入”和“弹出”操作类)的实现细节可以在不影响系统其余部分的情况下发生改变或被替换。

高级语言的工作是生成便于程序员使用的视图。没有统一的标准视图,其原因之一是编程语言种类较多。C++ 中的面向对象编程显示了另一种视图。

大多数编程语言都强调操作。在这些语言中,数据的存在是为了支持程序所需的操作。总之,数据和操作相比,显然次要得多。数据是原始的。内部数据类型为数不多,程序员很难生成自己的新数据类型。

C++ 和面向对象编程的出现改变了这一局面。C++ 提高了数据的重要性。C++ 中的主要操作就是生成自己的新数据类型(即类)和表达这些数据类型之间的相互作用。

要向这个方向发展,编程语言组织需要一些概念规范数据。我们考虑的规范化就是就是抽象数据类型的概念。抽象数据类型如同 10 多年前的结构化编程一样,备受关注,抽象数据类型不能代替结构化编程,只提供了其他规范,可以进一步改善程序开发过程。

什么是抽象数据类型呢?思考一下内部类型 `int`,它是数学中的整数,但是 `int` 在计算机中并不完全是数学中的整数,计算机中的 `int` 长度非常有限的。例如,32 位机器上的 `int` 只限于  $-20$  亿 ~  $+20$  亿之间。如果计算结果超出这个范围,则会发生溢出错误,机器会以其相关的方式响应,可能产生不易察觉的错误结果,而数学中的整数则不然。因此,计算机中的 `int` 概念实际上只是实际整数的一个近似值,`double` 也如此。

`char` 同样也是个近似,`char` 值通常是 8 位模式的 0 和 1,这些模式与所表示的字符(如大写字母 Z、小写字母 z、美元符号 \$、数字 5 等等)完全不同。`Char` 类型的值在大多数计算机上都是很有限的。7 位 ASCII 字符集只提供 128 个不同字符值。这显然不足以表达中文、日文等需要成千上万个字符的语言。

由此可见,即使是 C++ 之类的编程语言提供的内部数据类型,实际上也只是实际生活中概念和行为的近似或模型。前面我们一直在用 `int`,现在则有了全新的概念。`int`,`double`,`char` 之类的类型都是抽象数据类型。它们实际上是在计算机系统内,用可以接受的方式表示实际概念。

抽象数据概念实际上包含两个概念,即数据表达和该数据允许的操作。例如,`int` 的概念定义了 C++ 中的加、减、乘、除和求模操作,除数为 0 时则未定义,这种操作与机器参数有关,如计算机系统的定长字符长度。另一个例子是负整数概念,其运算和数据表达是可应用的,负整数的平方根则没有定义。C++ 中程序员用类实现抽象数据类型。我们将在第 12 章生成自己的堆栈类,并在第 20 章介绍 `stack` 标准库类。

### 7.8.1 示例:数组抽象数据类型

第 4 章曾介绍过数组。数组就是一个指针和一些内存空间。如果程序员小心谨慎,就可利用这些原始功能操作数组。数组还有很多非常好的操作,但是在 C++ 中没有提供。利用 C++ 类,程序员可以开发比“原始”数组更精彩的数组 ADT。数组类可以提供许多有用新功能,例如:

- 下标范围检查;
- 任意范围下标而不一定从 0 开始;
- 数组赋值;
- 数组比较;
- 数组输入/输出;
- 已知数组长度;

- 动态地扩展数组可以容纳更多的元素。

我们将在第8章生成自己的数组类,在第20章介绍 vector 标准库类。

**软件工程知识 7.15** 程序员可通过类机制生成新类型。这些新类型设计好之后,可以像使用内部类型一样方便地使用。所以我们说C++是可扩展的语言。尽管该语言可以随新类型扩展,但是基础语言本身是不会改变的。

在C++环境中生成的新类可以专属一个人、一个小组或一个公司。类也可以放入标准库中,以便更广泛的发行。尽管标准正逐渐浮出水面,但这不一定有必要上升为特定的标准。只有当充实而标准化的类库得到广泛应用时,C++的全部价值才能完全发挥出来。ANSI(美国国家标准协会)和ISO(国际标准化组织)已经开发了包含标准类库的C++标准版本。学习C++和面向对象编程的程序员可以利用丰富的库实现快速的、面向组件的软件开发。

### 7.8.2 示例:字符串抽象数据类型

C++是一种定义精炼的语言,只向程序员提供了建立各种系统的原始功能(可以把它视为生成工具的工具)。该语言为减轻编程负担而设计。C++适用于编程和系统编程,后者对程序性能的要求非常高。当然,C++内部数据类型中也可以包含字符串数据类型,但以语言设计为包括通过类生成和实现字符串抽象数据类型的机制。我们将在第8章开发字符串ADT。ANSI和ISO标准中有一个 string 类,详情参见第19章。

### 7.8.3 示例:队列抽象数据类型

我们每个人几乎都有过排队的经历。等待队称为队列。我们在超市排队结账,排队换煤气,排队乘公车,在高速公路上排队交费,学生注册时以及到食堂买饭时都要排队。计算机系统内部使用排队的情况也较多,鉴于此,我们要编写一个模拟排队的程序。

对于抽象数据类型,队列是一个很好的例子。队列向客户提供了容易理解的行为。客户一次把一件东西放进队列中,称为进队,然后从队列中一次一件地取出东西,称为出队。理论上讲,队列可以无限长。当然,实际上的队列是有限的。队列中的项目按先进先出顺序出列,第一个插入队列的项目第一个离开队列。

队列隐藏了如何跟踪队列中当前项目的内部数据表示式,为客户代码提供一组操作,如进队和出队。客户代码并不关心队列的实现细节,只要求队列正常操作。客户代码让一个新项目入队时,队列就接受这个项目并将其放入某种先进先出的数据结构。客户代码要从队列中取一个项目时,队列应从内部表中删除这个项目,并将其按先进先出的顺序向外传递(即队列的客户代码),下一个出队的项目就是队列中等待时间最长的项目。

队列ADT保证内部数据结构的完整性。客户代码不能直接操作这个数据结构,只有队列成员函数可以访问其内部数据。客户只能对数据表达进行合理的操作,ADT的 public 接口中不提供的操作将被ADT以适当的方式拒绝,即发送一个错误信息、中止执行或操作请求。

第15章将建立队列类,第20章将介绍 queue 标准库类。

## 7.9 容器类和迭代器

最常见的类型包括容器类(也称为集合类),即将类设计为保存一组对象集合。容器类通常提供插入、删除、查找、排序和测试类成员项目等操作确定该容器类是否是集合的成员。数组、堆栈、队列、树和链表都是容器类,第4章介绍了数组,第15章和第20章将介绍其他各种数据结构。

容器类经常与迭代对象(或简称迭代器)关联。迭代器是返回集合中下一个项目的对象(或对集合中下一个项目的某种操作)。编写迭代器之后,要取得类中下一个元素很简单,如同几个人共同阅读一本书可以在其中插入若干张书签一样,容器类可以有几个同时操作的迭代器。每个迭代器包含自己的位置信息。第20章将详细介绍容器和迭代器。

## 7.10 代理类

隐藏实现方法细节可以防止访问类的专属信息(包括 private 数据)和专属程序逻辑。向客户代码提供代理类,代理类只能访问类的 public 接口,这样既可以让客户使用类的服务又防止了客户访问类实现细节。

实现代理类需要几个步骤(如图7.10所示)。首先为要隐藏其 private 数据的类生成类定义和实现文件。图7.10所示程序中,第10~12行展示了我们的示例类,名为 Implementation。第13~41行展示了 Interface 代理类,第42~61行展示了测试程序和输出结果。

```

1 //Fig. 7.10: implementation.h
2 //Header file for class Implementation
3
4 class Implementation {
5     public:
6         Implementation( int v ) { value = v; }
7         void setValue( int v ) { value = v; }
8         int getValue() const { return value; }
9
10    private:
11        int value;
12 };

```

图 7.10 代理类的实现方法——implementation. h

```

13 //Fig. 7.10: interface.h
14 //Header file for interface.cpp
15 class Implementation; //forward class declaration
16
17 class Interface {
18     public:
19         Interface( int );
20         void setValue( int ); //same public interface as
21         int getValue() const; //class Implementation
22         ~Interface();

```

```

23 private;
24     Implementation *ptr; //requires previous
25                           //forward declaration
26 };

```

图 7.10 代理类的实现方法——implementation.h

```

27 //Fig. 7.10; interface.cpp
28 //Definition of class Interface
29 #include "interface.h"
30 #include "implementation.h"
31
32 Interface::Interface( int v )
33     : ptr( new Implementation( v ) ) {}
34
35 //call Implementation's setValue function
36 void Interface::setValue( int v ) { ptr ->setValue( v ); }
37
38 //call Implementation's getValue function
39 int Interface::getValue() const { return ptr ->getValue(); }
40
41 Interface::~~Interface() { delete ptr; }

```

图 7.10 代理类的实现方法——implementation.cpp

```

42 //Fig. 7.10; fig07_10.cpp
43 //Hiding a class private data with a proxy class.
44 #include <iostream>
45
46 using std::cout;
47 using std::endl;
48
49 #include "interface.h"
50
51 int main()
52 {
53     Interface i( 5 );
54
55     cout << "Interface contains: " << i.getValue()
56          << " before setValue" << endl;
57     i.setValue( 10 );
58     cout << "Interface contains: " << i.getValue()
59          << " after setValue" << endl;
60     return 0;
61 }

```

输出结果:

```

Interface contains: 5 before setVal
Interface contains: 10 after setVal

```

图 7.10 代理类的实现方法——fig07\_10.cpp

Implementation 类提供了一个 private 数据成员 value(即要对客户代码隐藏的数据),一个初始化 value 的构造函数以及函数 setValue 和 getValue。

我们用 Implementation 类的同一个 public 接口生成代理类的定义。代理类惟一的 private 成员是 Implementation 类对象的指针。利用指针可以对客户代码隐藏 Implementation 类的实现细节。

Interface 类是 Implementation 类的代理类。注意, Interface 类中提到 Implementation 类时只有指针声明(第 24 行)。类定义(如 Interface 类)只使用另一个类(如 Implementation 类)的指针时,另一个类的头文件(通常显示该类的 private 数据)不需要用 #include 包含在内。只须在文件使用某种数据类型之前用正向类声明将另一个类声明为一种数据类型即可(第 15 行)。

实现文件包含 Interface 代理类的成员函数,这是惟一包含 Implementation 类所在头文件 implementation.h 的文件。文件 interface.cpp 以预处理对象文件形式随同头文件 interface.h 一起提供给客户代码,该头文件包含代理类提供服务的函数原型。由于文件 interface.cpp 只以已编译对象文件形式提供给客户代码,因此客户代码无法看到代理类与专属类之间的交互。

图 7.10 中的程序最后测试了 Interface 类。注意,main 中只包含 Interface 类的头文件,没有提到 Implementation 类。因此,客户代码根本不会知道 Implementation 类的 private 数据,客户代码也不能依赖 Implementation 代码。

## 7.11 【可选案例分析】对象思想:为电梯模拟程序中的类编写程序

第 2~5 章的“对象思想”小节,我们设计了电梯模拟程序,第 6 章开始了用 C++ 实现电梯模拟程序的编程。第 7 章介绍了实现可投入使用的完整电梯模拟程序所需的其他技术,包括动态对象管理技术,用 new 和 delete 生成和删除对象。还介绍了合成,可在一个类中包含其他类对象成员。通过合成可以建立 Building 类,包含 Scheduler 对象、Clock 对象、Elevator 对象和两个 Floor 对象,并且每一个 Elevator 类可以包含每一个 ElevatorButton, Door 和 Bell 类的一个对象;Floor 类可以包含 FloorButton 和 Light 对象。还介绍了如何使用 static 类成员、常量类成员以及构造函数中的成员初始化语法。这里,我们将用以上这些技术继续用 C++ 实现我们的电梯模拟程序。本节最后还提供了完整的电梯模拟 C++ 程序(约有 1 000 行代码)和一个详细代码预排工作。第 9 章“对象思想”小节中,我们在电梯模拟程序中采用了继承从而完成了我们的电梯模拟案例分析,我们只提供了实现继承的额外的 C++ 代码。

### 7.11.1 电梯模拟系统实现方法概述

电梯模拟系统由 Building 类的对象控制,它包含两个 Floor 类对象、一个 Elevator, Clock 和 Scheduler 类对象。第 2 章“对象思想”小节的 UML 类图表(图 2.44)演示了这种组成关系。时钟只根据当前时间的每一秒而变化并通过大楼每次递增一秒。时间表负责安排人到达每一层楼的时间。

时钟每滴答一秒,大楼就根据当前时间(通过 Scheduler 类的 processTime 成员函数)更

新时间表。时间表检查当前时间,安排人到达每个楼层的下一个时间。如果安排人到达某个楼层时,时间表需要通过调用 Floor 类的 isOccupied 成员函数检查该楼层是否为空。如果调用返回 true,则表明当前有人在该楼层,时间表将调用 delayArrival 函数将到达该楼层的下一个时间延迟一秒。

如果楼层为空的(即调用返回 false),时间表将生成一个新的 Person 类对象,这个人将步行到适当的楼层,然后调用 FloorButton 类的 pressButton 成员函数。接着楼层按钮调用 Elevator 类的 summonElevator 方法。

时钟每嘀答一次,大楼就会更新电梯的当前时间。接到更新后的时间,电梯首先会检查电梯的当前状态(即“运行”或“停止”)。如果电梯正在两个楼层之间运行,但是并没有预先设置电梯应在此时到达一个楼层,电梯就只会在屏幕上输出运行的方向。如果电梯正在运行,并且当前时间和安排好的下次到达楼层的时间相吻合,电梯将停止运行,重新按电梯按钮,电梯铃响并通知楼层电梯已经到达该楼层(通过调用 Floor 类的 elevatorArrived 成员函数)。楼层做出回应,重新按楼层按钮并打开楼层灯。然后电梯打开门,允许电梯里的人离开,楼层上的人进入电梯。接着电梯关上门并确定下一次将在哪一个楼层停。如果其他楼层有人需要乘电梯,电梯将移向该楼层。

如果电梯从大楼接到更新的时间时处于停止状态,电梯就需要确定哪一个楼层需要服务。如果当前楼层需要服务(即有人按电梯当前楼层的按钮),电梯铃会响,表示电梯已经到达随铃打开电梯门。站在楼层等候的人进入电梯,按电梯的按钮,使电梯开始移向另外的楼层移动。如果其他楼层需要服务(即有人按了其他楼层的按钮),则电梯会移向该楼层移动。

### 7.11.2 电梯模拟程序的实现方法

前面的“对象思想”小节收集了很多关于电梯系统的信息。用这些信息生成一个电梯模拟系统的面向对象设计,并用 UML 演示这个设计。我们已经学习了所有实现一个实用性模拟系统所需的 C++ 面向对象编程技术。本节稍后将介绍 C++ 实现方法和详细的代码预排工作。

驱动程序(见图 7.1)首先提示用户输入模拟系统应该运行的时间长度(第 15 行和第 16 行)。第 17 行的 cin.ignore 调用指示 cin 流忽略用户在运行时整数后输入的返回字符。这样将从输入流中删除返回字符。然后驱动程序生成 building 对象(第 19 行)并调用该对象的 runSimulation 成员函数,将用户指定的持续时间作为参数进行传递(第 23 行)。驱动程序还打印出信息,提醒用户模拟程序何时开始(第 21 行),何时结束(第 24 行)。

```

1 //Figure 7.11
2 //Driver for the simulation.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "building.h"
10
11 int main()
```



```

12 |
13   int duration;           //length of simulation in seconds
14
15   cout << "Enter run time:";
16   cin >> duration;
17   cin.ignore();           //ignore return char
18
19   Building building;       //create the building
20
21   cout << endl << " * * * ELEVATOR SIMULATION BEGINS * * *"
22       << endl << endl;
23   building.runSimulation(duration); //start simulation
24   cout << " * * * ELEVATOR SIMULATION ENDS * * *" << endl;
25
26   return 0;
27 }

```

图 7.11 电梯模拟程序的驱动程序

根据类图表(如图 2.44 所示), Building 类由其他几个类组成。图 7.12 中的 Building 头文件显示了这种组成(第 46 ~ 50 行)。Building 类由两个 Floor 对象(floor1 和 floor2)、一个 Elevator 对象(elevator)、一个 Clock 对象(Clock)和一个 Scheduler 对象(scheduler)组成。图 7.13 演示了 Building 类的实现文件。构造函数在第 64 行至第 69 行。在成员初始化列表(第 65 ~ 68 行)中,组成 Building 类的许多对象的构造函数用适当的参数引用。FLOOR1 和 FLOOR2(第 65 行和第 66 行)在 Floor 类(第 821 行和第 822 行)中被定义为常量。

```

28 //building.h
29 //Definition for class Building.
30 #ifndef BUILDING_H
31 #define BUILDING_H
32
33 #include "elevator.h"
34 #include "floor.h"
35 #include "clock.h"
36 #include "scheduler.h"
37
38 class Building {
39
40 public:
41   Building();           //constructor
42   ~Building();          //destructor
43   void runSimulation( int ); //run simulation for specified time
44
45 private:
46   Floor floor1;         //floor1 object
47   Floor floor2;         //floor2 object
48   Elevator elevator;    //elevator object
49   Clock clock;          //clock object
50   Scheduler scheduler;  //scheduler object
51 };

```

```

52
53 #endif //BUILDING_H

```

图 7.12 Building 类的头文件

```

54 //building.cpp
55 //Member function definitions for class Building.
56 #include <iostream>
57
58 using std::cout;
59 using std::cin;
60 using std::endl;
61
62 #include "building.h"
63
64 Building::Building() //constructor
65     : floor1( Floor::FLOOR1, elevator ),
66       floor2( Floor::FLOOR2, elevator ),
67       elevator( floor1, floor2 ),
68       scheduler( floor1, floor2 )
69 { cout << "building created" << endl; }
70
71 Building::~Building() //destructor
72 { cout << "building destroyed" << endl; }
73
74 //control the simulation
75 void Building::runSimulation( int totalTime )
76 {
77     int currentTime = 0;
78
79     while ( currentTime < totalTime ) {
80         clock.tick();
81         currentTime = clock.getTime();
82         cout << "TIME: " << currentTime << endl;
83         scheduler.processTime( currentTime );
84         elevator.processTime( currentTime );
85         cin.get(); //stop each second for user to view output
86     }
87 }

```

图 7.13 Building 类的实现文件

Building 类的主要功能体现在其 runSimulation 成员函数(第 74 ~ 87 行)中,该成员函数将一直循环直到传递完指定的时间为止。在每个迭代器上,building 指示 Clock 向 Clock 发送 tick 信息(第 80 行),以便一次递增一秒。然后 building 通过调用 getTime 成员函数(第 81 行)从 clock 重新得到时间。接着通过 processTime 和 currentTime 分别向 scheduler 和 elevator(第 83 行和第 84 行)发送信息。最后添加 cin.get 调用(第 85 行)允许用户停止临时输出滚屏,按回车键继续屏幕输出之前,要观察模拟程序输出的下一个模拟时间的秒数。

Clock 类较为简单,它不是由其他任何类组成的。图 7.14 中显示了 Clock 类的头文件,图 7.15 显示了它的实现方法。Clock 类的对象可以通过 tick 成员函数(函数原型在第 98

行,实现方法在第 122 行和第 123 行)接收信息,递增 time。通过第 99 行、第 125 行和第 126 行的 getTime 成员函数,其他对象也可以访问当前时间。注意,getTime 函数是常量类型。

```

88 //clock.h
89 //Definition for class Clock.
90 #ifndef CLOCK_H
91 #define CLOCK_H
92
93 class Clock {
94
95 public:
96     Clock();           //constructor
97     ~Clock();          //destructor
98     void tick();        //increment clock by one second
99     int getTime() const; //returns clock's current time
100
101 private:
102     int time;          //clock's time
103 };
104
105 #endif //CLOCK_H

```

图 7.14 Clock 类的头文件

```

106 //clock.cpp
107 //Member function definitions for class Clock.
108 #include <iostream>
109
110 using std::cout;
111 using std::endl;
112
113 #include "clock.h"
114
115 Clock::Clock()           //constructor
116     : time( 0 )
117 { cout << "clock created" << endl; }
118
119 Clock::~~Clock()         //destructor
120 { cout << "clock destroyed" << endl; }
121
122 void Clock::tick()        //increment time by 1
123 { time ++; }
124
125 int Clock::getTime() const //return current time
126 { return time; }

```

图 7.15 Clock 类的实现文件

Scheduler 类(参见图 7.16)负责随机生成 Person 类的对象并将生成的对象放入适当的楼层。public 接口列出了 processTime 成员函数,该函数取当前时间作为参数(第 139 行)。头文件也列出了几个 private 工具函数(稍后将介绍),可以执行 processTime 成员函数要求的

任务。

```

127 //scheduler.h
128 //defintion for class Scheduler
129 #ifndef SCHEDULER_H
130 #define SCHEDULER_H
131
132 class Floor;           //forward declaration
133
134 class Scheduler {
135
136 public:
137     Scheduler( Floor &, Floor & ); //constructor
138     ~Scheduler();                 //destructor
139     void processTime( int );      //set scheduler's time
140
141 private:
142     //schedule arrival to a floor
143     void scheduleTime( const Floor & );
144
145     //delay arrival to a floor
146     void delayTime( const Floor & );
147
148     //create new person; place on floor
149     void createNewPerson( Floor & );
150
151     //handle person arrival on a floor
152     void handleArrivals( Floor &, int );
153
154     int currentClockTime;
155
156     Floor &floor1Ref;
157     Floor &floor2Ref;
158
159     int floor1ArrivalTime;
160     int floor2ArrivalTime;
161 };
162
163 #endif //SCHEDULER_H

```

图 7.16 Scheduler 类的头文件

图 7.17 列出了 Scheduler 类的实现文件。ProcessTime 成员函数(第 222 ~ 232 行)将大多数责任委托给类中比较小的工具函数。Scheduler 类的构造函数(第 178 行和第 189 行)首先用伪随机数生成器发出当前真实世界时间基础上的数字(第 183 行)。这使随机数生成器产生了一系列每次执行程序时的不同数字。接着 Scheduler 类对每个楼层调用 scheduleTime 工具函数(第 187 行和第 188 行)。这个函数计算第一个到达每个楼层的 Person 对象的伪随机到达时间(本例中,第 5 ~ 20 行包含随机数)。

```
164 //scheduler.cpp
165 //Member function definitions for class Scheduler.
166 #include <iostream>
167
168 using std::cout;
169 using std::endl;
170
171 #include <cstdlib>
172 #include <ctime>
173
174 #include "scheduler.h"
175 #include "floor.h"
176 #include "person.h"
177
178 //constructor
179 Scheduler::Scheduler( Floor &firstFloor, Floor &secondFloor )
180 : currentClockTime( 0 ), floor1Ref( firstFloor ),
181   floor2Ref( secondFloor )
182 {
183     srand( time( 0 ) ); //seed random number generator
184     cout << "scheduler created" << endl;
185
186     //schedule first arrivals for floor 1 and floor 2
187     scheduleTime( floor1Ref );
188     scheduleTime( floor2Ref );
189 }
190
191 Scheduler::~Scheduler() //destructor
192 { cout << "scheduler destroyed" << endl; >
193
194 //schedule arrival on a floor
195 void Scheduler::scheduleTime( const Floor &floor )
196 {
197     int floorNumber = floor.getNumber();
198     int arrivalTime = currentClockTime + ( 5 + rand() % 16 );
199
200     floorNumber == Floor::FLOOR1 ?
201         floor1ArrivalTime = arrivalTime :
202         floor2ArrivalTime = arrivalTime;
203
204     cout << "(scheduler schedules next person for floor "
205           << floorNumber << " at time " << arrivalTime << ' '
206           << endl;
207 }
208
209 //reschedule an arrival on a floor
210 void Scheduler::delayTime( const Floor &floor )
211 {
212     int floorNumber = floor.getNumber();
213
214     int arrivalTime = ( floorNumber == Floor::FLOOR1 ) ?
```

```
215     ++floor1ArrivalTime : ++floor2ArrivalTime;
216
217     cout << "(scheduler delays next person for floor "
218           << floorNumber << " until time " << arrivalTime << ')'
219           << endl;
220 }
221
222 //give time to scheduler
223 void Scheduler::processTime( int time )
224 {
225     currentClockTime = time; //record time
226
227     //handle arrivals on floor 1
228     handleArrivals( floor1Ref, currentClockTime );
229
230     //handle arrivals on floor 2
231     handleArrivals( floor2Ref, currentClockTime );
232 }
233
234 //create new person and place it on specified floor
235 void Scheduler::createNewPerson( Floor &floor )
236 {
237     int destinationFloor =
238         floor.getNumber() == Floor::FLOOR1 ?
239         Floor::FLOOR2 : Floor::FLOOR1;
240
241     //create new person
242     Person *newPersonPtr = new Person( destinationFloor );
243
244     cout << "scheduler creates person "
245           << newPersonPtr ->getID() << endl;
246
247     //place person on proper floor
248     newPersonPtr ->stepOntoFloor( floor );
249
250     scheduleTime( floor ); //schedule next arrival
251 }
252
253 //handle arrivals for a specified floor
254 void Scheduler::handleArrivals( Floor &floor, int time )
255 {
256     int floorNumber = floor.getNumber();
257
258     int arrivalTime = ( floorNumber == Floor::FLOOR1 ) ?
259         floor1ArrivalTime : floor2ArrivalTime;
260
261     if ( arrivalTime == time ) {
262
263         if ( floor.isOccupied() ) //see if floor occupied
264             delayTime( floor );
265         else
```

```

266     createNewPerson( floor );
267     |
268     {

```

图 7.17 Scheduler 类的实现文件

在电梯模拟系统中, building 通过 scheduler 的 processTime 成员函数随着当前时间更新 scheduler 的每一秒(第 222 行和第 232 行)。图 4.27 所示的顺序图表模拟对信息反应出的行为的顺序, 实现方法表现出这种模拟。调用 processTime 时, scheduler 对每个楼层调用 handleArrivals 工具函数(第 228 ~ 231 行)。该工具函数将当前 time(由 building 提供)作为安排的到达楼层的下一个时间(第 261 行)。如果当前时间与到达楼层的时间相同, 并且楼层被占用(第 263 行), scheduler 调用 delayTime 工具函数将下一个安排的到达时间延迟一秒(第 264 行)。如果楼层未被占用, scheduler 调用工具函数 creatNewPerson(第 266 行)通过使用 new 操作符生成 Person 类的一个新对象(第 242 行)。然后 scheduler 向 Person 类的新对象发送 stepOntoFloor 信息(第 248 行)。当人已经步行到楼层, scheduler 通过调用工具函数 scheduleTime 为楼层上的人计算下一个到达时间(第 250 行)。

我们已经检验过组成模拟系统控制器部分所有类的实现方法。现在要检验组成模拟系统主体的类。Bell 类和 Clock 类一样, 不是由其他对象组成的。图 7.18 头文件中定义的类 Bell 的 public 接口包含构造函数、析构函数和 ringBell 成员函数。这些函数的实现(图 7.19 中的第 292 行和第 293 行、第 295 行和第 296 行以及第 298 行和第 299 行)只是在屏幕输出信息。

```

269 //bell.h
270 //Definition for class Bell.
271 #ifndef BELL_H
272 #define BELL_H
273
274 class Bell {
275
276 public:
277     Bell();           //constructor
278     ~Bell();          //destructor
279     void ringBell() const; //ring the bell
280 };
281
282 #endif //BELL_H

```

图 7.18 Bell 类的头文件

```

283 //bell.cpp
284 //Member function definitions for class Bell.
285 #include <iostream>
286
287 using std::cout;
288 using std::endl;
289
290 #include "bell.h"
291

```

```

292 Bell::Bell() //constructor
293 { cout << "bell created" << endl; }
294
295 Bell::~~Bell() //destructor
296 | cout << "bell destroyed" << endl; |
297
298 void Bell::ringBell() const //ring bell
299 { cout << "elevator rings its bell" << endl; >

```

图 7.19 Bell 类的实现文件

Light 类(见图 7.20 和图 7.21)在其 public 接口中显示了除构造函数和析构函数之外的另两个成员函数。TurnOn 成员函数只是通过将 on 数据成员设置为 true 将灯打开(第 335 ~ 339 行),turnoff 成员函数通过将 on 数据成员设置为 false 的方式,关闭了灯(第 341 ~ 345 行)。

```

300 //light.h
301 //Definition for class Light.
302 #ifndef LIGHT_H
303 #define LIGHT_H
304
305 class Light {
306
307 public:
308     Light( const char * ); //constructor
309     ~Light(); //destructor
310     void turnOn(); //turns light on
311     void turnOff(); //turns light off
312
313 private:
314     bool on; //true if on; false if off
315     const char *name; //which floor the light is on
316 };
317
318 #endif //LIGHT_H

```

图 7.20 Light 类的头文件

```

319 //light.cpp
320 //Member function definitions for class Light.
321 #include <iostream>
322
323 using std::cout;
324 using std::endl;
325
326 #include "light.h"
327
328 Light::Light( const char *string ) //constructor
329 : on( false ), name( string )
330 { cout << name << " light created" << endl; }
331
332 Light::~~Light() //destructor

```



```

333 | cout << name << " light destroyed" << endl; |
334
335 void Light::turnOn() //turn light on
336 |
337     on = true;
338     cout << name << " turns on its light" << endl;
339 |
340
341 void Light::turnOff() //turn light off
342 |
343     on = false;
344     cout << name << " turns off its light" << endl;
345 |

```

图 7.21 Light 类的实现文件

Door 类(见图 7.22 和图 7.23)在我们的电梯模拟程序中较为重要。Door 对象显示电梯乘客将要离开,还显示在楼层等待的人进入 elevator。这些行为由 Door 类的 openDoor 成员函数来完成。注意,openDoor 成员函数取 4 个参数(第 361 行和第 362 行)。第一个参数是占用 elevator 的 Person 类对象,第二个参数是在楼层等待的 Person 类对象的指针,其余参数是 Floor 类的对象和 elevator 对象的引用。

```

346 //door.h
347 //Definition for class Door.
348 #ifndef DOOR_H
349 #define DOOR_H
350
351 class Person;           //forward declaration
352 class Floor;            //forward declaration
353 class Elevator;         //forward declaration
354
355 class Door {
356
357 public:
358     Door();              //constructor
359     ~Door();             //destructor
360
361     void openDoor( Person * const, Person * const,
362                  Floor &, Elevator & );
363     void closeDoor( const Floor & );
364
365 private:
366     bool open;           //open or closed
367 };
368
369 #endif //DOOR_H

```

图 7.22 Door 类的头文件

```

370 //door.cpp
371 //Member function definitions for class Door.
372 #include <iostream>

```

```

373
374 using std::cout;
375 using std::endl;
376
377 #include "door.h"
378 #include "person.h"
379 #include "floor.h"
380 #include "elevator.h"
381
382 Door::Door() //constructor
383     : open( false )
384 { cout << "door created" << endl; }
385
386 Door::~~Door() //destructor
387 { cout << "door destroyed" << endl; }
388
389 //open the door
390 void Door::openDoor( Person * const passengerPtr,
391                     Person * const nextPassengerPtr,
392                     Floor &currentFloor, Elevator &elevator )
393 {
394     if ( ! open ) {
395         open = true;
396
397         cout << "elevator opens its door on floor "
398              << currentFloor.getNumber() << endl;
399
400         if ( passengerPtr != 0 ) {
401             passengerPtr ->exitElevator( currentFloor, elevator );
402             delete passengerPtr; //passenger leaves simulation
403         }
404
405         if ( nextPassengerPtr != 0 )
406             nextPassengerPtr ->enterElevator(
407                 elevator, currentFloor );
408     }
409 }
410
411 //close the door
412 void Door::closeDoor( const Floor &currentFloor )
413 {
414     if ( open ) {
415         open = false;
416         cout << "elevator closes its door on floor "
417              << currentFloor.getNumber() << endl;
418     }
419 }

```

图 7.23 Door 类的实现文件

Door 类是 Elevator 类的合成对象,要实现这个合成,Elevator 类的头文件必须包含语句

```
#include "door.h"
```

Door 类使用了 Elevator 类对象的引用(第 362 行)。要声明 Elevator 类以使 Door 类可以使用这个引用,需要在 Door 类的头文件加入语句

```
#include "elevator.h"
```

如此一来,Elevator 类的头文件要包含 Door 类的头文件,反之亦然。处理器不但不能处理#include这样的指令,还会因为这个循环包容问题产生致命的错误。

要避免这个问题,应将 Elevator 类的提前声明放入 Door 类的头文件(第 353 行)。这条提前声明通知处理器我们要引用文件中的 Elevator 类对象,但是 Elevator 类的定义放在文件之外。注意,我们还对 Person 类和 Floor 类做了提前声明(第 351 行和第 352 行),因此可以对 openDoor 成员函数使用这些类的原型。

图 7.23 列出了 Door 类的实现文件。第 378 ~ 380 行包含了 Person、Floor 和 Elevator 类的头文件。这些#include说明符合头文件中的提前声明,并且这些头文件包含可以引用这些类的适当成员函数的指定函数原型。

调用 openDoor 成员函数时(第 389 ~ 409 行),首先检查 door 是否尚未打开。Door 检查指向电梯中乘客的指针(passengerPtr)不是 0(第 400 行)。如果指针不是 0,说明电梯中有一个需要离开的人。通过 exitElevator 信息通知人准备离开电梯(第 401 行)。Door 通过 delete 操作符删除乘坐 elevator 的 Person 类对象(第 402 行)。

当乘客离开电梯,电梯门检查在楼层等候之人的指针(nextPassengerPtr),确认该指针不是 0(第 405 行)。如果该指针不是 0(即有一位准备进入电梯的乘客),通过 Person 类的 enterElevator 成员函数允许乘客进入电梯(第 406 行和第 407 行)。Door 类的 closeDoor 成员函数(第 412 ~ 419 行)只检查电梯门是否开着,如果门还开着就将其关闭。

系统中的人用 ElevatorButton 类对象(参见图 7.24 和 7.25)开始将电梯移向其他楼层。PressButton 成员函数(第 460 ~ 466 行)电梯按钮的 pressed 属性设置为 true,然后将 prepareToLeave 信息发送到 elevator。ResetButton 成员函数将 pressed 属性设置为 false。

```
420 //elevatorButton.h
421 //Definition for class ElevatorButton.
422 #ifndef ELEVATORBUTTON_H
423 #define ELEVATORBUTTON_H
424
425 class Elevator;           //forward declaration
426
427 class ElevatorButton {
428
429 public:
430     ElevatorButton( Elevator & ); //constructor
431     ~ElevatorButton();           //destructor
432
433     void pressButton();           //press the button
434     void resetButton();           //reset the button
435
436 private:
437     bool pressed;                //state of button
```

```

438 Elevator &elevatorRef;          //reference to button's elevator
439 |;
440
441 #endif //ELEVATORBUTTON_H

```

图 7.24 ElevatorButton 类的头文件

```

442 //elevatorButton.cpp:
443 //Member function definitions for class ElevatorButton.
444 #include <iostream>
445
446 using std::cout;
447 using std::endl;
448
449 #include "elevatorButton.h"
450 #include "elevator.h"
451
452 //constructor
453 ElevatorButton::ElevatorButton( Elevator &elevatorHandle )
454     : pressed( false ), elevatorRef( elevatorHandle )
455 { cout << "elevator button created" << endl; |
456
457 ElevatorButton::~ElevatorButton() //destructor
458 { cout << "elevator button destroyed" << endl; |
459
460 void ElevatorButton::pressButton() //press the button
461 |
462     pressed = true;
463     cout << "elevator button tells elevator to prepare to leave"
464         << endl;
465     elevatorRef.prepareToLeave( true );
466 |
467
468 void ElevatorButton::resetButton() //reset the button
469 { pressed = false; |

```

图 7.25 ElevatorButton 类的实现文件

FloorButton 类(参见图 7.26 和图 7.27)通过 public 接口显示了与 ElevatorButton 类相同的成员函数。public pressButton 成员函数通过 summonElevator 信息召唤 elevator。调用 resetButton 成员函数可重新设置楼层按钮。

```

470 //floorButton.h
471 //Definition for class FloorButton.
472 #ifndef FLOORBUTTON_H
473 #define FLOORBUTTON_H
474
475 class Elevator;          //forward declaration
476
477 class FloorButton |
478
479 public;

```

```

480 FloorButton( const int, Elevator & ); //constructor
481 ~FloorButton(); //destructor
482
483 void pressButton(); //press the button
484 void resetButton(); //reset the button
485
486 private:
487     const int floorNumber; //number of the button's floor
488     bool pressed; //state of button
489
490     //reference to button's elevator
491     Elevator &elevatorRef;
492 };
493
494 #endif //FLOORBUTTON_H

```

图 7.26 FloorButton 类的头文件

```

495 //floorButton.cpp
496 //Member function definitions for class FloorButton.
497 #include <iostream>
498
499 using std::cout;
500 using std::endl;
501
502 #include "floorButton.h"
503 #include "elevator.h"
504
505 //constructor
506 FloorButton::FloorButton( const int number,
507                           Elevator &elevatorHandle )
508     : floorNumber( number ), pressed( false ),
509       elevatorRef( elevatorHandle )
510 {
511     cout << "floor " << floorNumber << " button created"
512          << endl;
513 }
514
515 FloorButton::~~FloorButton() //destructor
516 {
517     cout << "floor " << floorNumber << " button destroyed"
518          << endl;
519 }
520
521 //press the button
522 void FloorButton::pressButton()
523 {
524     pressed = true;
525     cout << "floor " << floorNumber
526          << " button summons elevator" << endl;
527     elevatorRef.summonElevator( floorNumber );
528 }

```

```

529
530 //reset the button
531 void FloorButton::resetButton()
532 { pressed = false; }

```

图 7.27 FloorButton 类的实现文件

Elevator 类的头文件(见图 7.28)是电梯模拟程序中最复杂的一部分。Elevator 在 public 接口中显示了 5 个成员函数(构造函数和析构函数除外)。ProcessTime 成员函数允许大楼向 elevator 发送更新的时钟 time。summonElevator 成员函数允许 Person 对象向 elevator 发送信息,要求 elevator 提供服务。成员函数 passengerEnters 和 passengerExit 使乘客进入和离开 elevator。prepareToLeave 成员函数使 elevator 在开始移向另一个楼层前执行需要完成的任何任务。elevatorButton 对象声明为 public,因此 Person 类对象可以直接访问 elevatorButton。通常人不与铃或门接触(电梯技术人员除外)。因此,在类定义的 private 部分声明 bell 和 door 对象。

```

533 //elevator.h
534 //Definition for class Elevator.
535 #ifndef ELEVATOR_H
536 #define ELEVATOR_H
537
538 #include "elevatorButton.h"
539 #include "door.h"
540 #include "bell.h"
541
542 class Floor;           //forward declaration
543 class Person;          //forward declaration
544
545 class Elevator {
546
547 public:
548     Elevator( Floor &, Floor & );    //constructor
549     ~Elevator();                    //destructor
550     void summonElevator( int );      //request to service a floor
551     void prepareToLeave( bool );      //prepare to leave
552     void processTime( int );         //give time to elevator
553     void passengerEnters( Person * const ); //board a passenger
554     void passengerExits();           //exit a passenger
555     ElevatorButton elevatorButton; //note public object
556
557 private:
558     void processPossibleArrival();
559     void processPossibleDeparture();
560     void arriveAtFloor( Floor & );
561     void move();
562
563     //time to move between floors
564     static const int ELEVATOR_TRAVEL_TIME;
565     static const int UP;           //UP direction
566     static const int DOWN;        //DOWN direction

```

```

567
568   int currentBuildingClockTime; //current time
569   bool moving;                  //elevator state
570   int direction;                //current direction
571   int currentFloor;             //current location
572   int arrivalTime;              //time to arrive at a floor
573   bool floor1NeedsService;      //floor1 service flag
574   bool floor2NeedsService;      //floor2 service flag
575
576   Floor &floor1Ref;              //reference to floor1
577   Floor &floor2Ref;              //reference to floor2
578   Person *passengerPtr;         //pointer to current passenger
579
580   Door door;                    //door object
581   Bell bell;                    //bell object
582 };
583
584 #endif //ELEVATOR_H

```

图 7.28 Elevator 类的头文件

第 558 ~ 561 行包含工具函数。Elevator 类定义了一系列 `private` 静态常量值(第 564 ~ 566 行)。这些值指定为 `static` 类型,因为它们包含 Elevator 类所有对象的使用信息。这些值不能修改,所以被声明为常量。

Elevator 头文件的第 568 ~ 581 行包含额外的 `private` 数据成员。注意,对 Floor 类的每个对象都提供了引用句柄(第 576 行和第 577 行),但是对乘客对象使用指针(第 578 行)。对乘客对象使用指针是因为每次 Person 类对象进入或离开 elevator 时,句柄都会发生变化。我们更提倡引用 Floor 对象的句柄。

我们使用 UML 模拟了 Elevator 类的很多行为和动作(见图 3.31、图 3.32 和图 5.37),Elevator 类的代码(参见图 7.29)的实现方法信息包含在这些模式中。Elevator 构造函数有一个扩展的成员初始化值列表(第 602 ~ 607 行)。记住,ElevatorButton 类定义(参见图 7.24)中,类对象需要取 Elevator 类对象构造函数的句柄作为参数。在成员初始化列表中复引用 elevator 的 `this` 指针(第 602 行)即可提供该句柄。一些编译器会对这条语句发出警告消息,因此 elevator 对象尚未完全初始化。

```

585 //elevator.cpp
586 //Member function definitions for class Elevator.
587 #include <iostream>
588
589 using std::cout;
590 using std::endl;
591
592 #include "elevator.h"
593 #include "person.h"
594 #include "floor.h"
595
596 const int Elevator::ELEVATOR_TRAVEL_TIME = 5;
597 const int Elevator::UP = 0;

```

```

598 const int Elevator::DOWN = 1;
599
600 //constructor
601 Elevator::Elevator( Floor &firstFloor, Floor &secondFloor )
602     : elevatorButton( *this ), currentBuildingClockTime( 0 ),
603     moving( false ), direction( UP ),
604     currentFloor( Floor::FLOOR1 ), arrivalTime( 0 ),
605     floor1NeedsService( false ), floor2NeedsService( false ),
606     floor1Ref( firstFloor ), floor2Ref( secondFloor ),
607     passengerPtr( 0 )
608 { cout << "elevator created" << endl; >
609
610 Elevator::~~Elevator() //destructor
611 { cout << "elevator destroyed" << endl; }
612
613 //give time to elevator
614 void Elevator::processTime( int time )
615 {
616     currentBuildingClockTime = time;
617
618     if ( moving )
619         processPossibleArrival();
620     else
621         processPossibleDeparture();
622
623     if ( ! moving )
624         cout << "elevator at rest on floor "
625             << currentFloor << endl;
626 }
627
628 //when elevator is moving, determine if it should stop
629 void Elevator::processPossibleArrival()
630 {
631     //if elevator arrives at destination floor
632     if ( currentBuildingClockTime == arrivalTime ) {
633
634         currentFloor = //update current floor
635             ( currentFloor == Floor::FLOOR1 ?
636               Floor::FLOOR2 : Floor::FLOOR1 );
637
638         direction = //update direction
639             ( currentFloor == Floor::FLOOR1 ? UP : DOWN );
640
641         cout << "elevator arrives on floor "
642             << currentFloor << endl;
643
644         arriveAtFloor( currentFloor == Floor::FLOOR1 ?
645             floor1Ref : floor2Ref );
646
647         return;
648     }

```



```
649
650 //elevator is moving
651 cout << "elevator moving "
652     << ( direction == UP ? "up" : "down" ) << endl;
653 |
654
655 //determine if elevator should move
656 void Elevator::processPossibleDeparture()
657 |
658 //this floor needs service?
659 bool currentFloorNeedsService =
660     currentFloor == Floor::FLOOR1 ?
661     floor1NeedsService : floor2NeedsService;
662
663 //other floor needs service?
664 bool otherFloorNeedsService =
665     currentFloor == Floor::FLOOR1 ?
666     floor2NeedsService : floor1NeedsService;
667
668 //service this floor (if needed)
669 if ( currentFloorNeedsService ) {
670     arriveAtFloor( currentFloor == Floor::FLOOR1 ?
671         floor1Ref : floor2Ref );
672
673     return;
674 |
675
676 //service other floor (if needed)
677 else prepareToLeave( otherFloorNeedsService );
678 |
679
680 //arrive at a particular floor
681 void Elevator::arriveAtFloor( Floor& arrivalFloor )
682 {
683     moving = false; //reset state
684
685     cout << "elevator resets its button" << endl;
686     elevatorButton.resetButton();
687
688     bell.ringBell();
689
690     //notify floor that elevator has arrived
691     Person *floorPersonPtr = arrivalFloor.elevatorArrived();
692
693     door.openDoor( passengerPtr, floorPersonPtr,
694         arrivalFloor, *this );
695
696 //this floor needs service?
697 bool currentFloorNeedsService =
698     currentFloor == Floor::FLOOR1 ?
699     floor1NeedsService : floor2NeedsService;
```

```
700
701 //other floor needs service?
702 bool otherFloorNeedsService =
703     currentFloor == Floor::FLOOR1 ?
704         floor2NeedsService ; floor1NeedsService;
705
706 //if this floor does not need service
707 //prepare to leave for the other floor
708 if ( ! currentFloorNeedsService )
709     prepareToLeave( otherFloorNeedsService );
710 else //otherwise, reset service flag
711     currentFloor == Floor::FLOOR1 ?
712         floor1NeedsService = false; floor2NeedsService = false;
713 |
714
715 //request service from elevator
716 void Elevator::summonElevator( int floor )
717 |
718     //set appropriate servicing flag
719     floor == Floor::FLOOR1 ?
720         floor1NeedsService = true ; floor2NeedsService = true;
721 |
722
723 //accept a passenger
724 void Elevator::passengerEnters( Person * const personPtr )
725 |
726     //board passenger
727     passengerPtr = personPtr;
728
729     cout << "person " << passengerPtr ->getID()
730         << " enters elevator from floor "
731         << currentFloor << endl;
732 |
733
734 //notify elevator that passenger is exiting
735 void Elevator::passengerExits() | passengerPtr = 0; |
736
737 //prepare to leave a floor
738 void Elevator::prepareToLeave( bool leaving )
739 |
740     Floor &thisFloor =
741         currentFloor == Floor::FLOOR1 ? floor1Ref : floor2Ref;
742
743     //notify floor that elevator may be leaving
744     thisFloor.elevatorLeaving();
745
746     door.closeDoor( thisFloor );
747
748     if ( leaving ) //leave, if necessary
749         move();
750 |
```

```

751
752 void Elevator::move() //go to a particular floor
753 {
754     moving = true; //change state
755
756     //schedule arrival time
757     arrivalTime = currentBuildingClockTime +
758         ELEVATOR_TRAVEL_TIME;
759
760     cout << "elevator begins moving "
761         << ( direction == DOWN ? "down " : "up ")
762         << "to floor "
763         << ( direction == DOWN ? '1' : '2' )
764         << " (arrives at time " << arrivalTime << ' ' )
765         << endl;
766 }

```

图 7.29 Elevator 类的实现文件

building 调用 Elevator 类的 processTime 成员函数(第 613 ~ 626 行),将当前模拟系统中的 time 作为参数传递。该成员函数随当前模拟系统时间更新 currentBuildingClockTime 数据成员(第 616 行),然后检查 motion 数据成员的值(第 618 行)。如果电梯正在移动,调用 processPossibleArrival 工具函数(第 619 行)。如果电梯没有移动,elevator 将调用 processPossibleDeparture 工具函数(第 621 行)。如果在确定电梯将到达当前楼层或移向其他楼层之后,电梯仍然没有移动,elevator 会输出一条信息表明电梯正在 currentFloor(第 623 ~ 625 行)休息。

processPossibleArrival 函数将 currentBuildingClockTime 作为计算过的 arrivalTime(第 632 行),以此确定 elevator 是否要停止移动。如果此时正是 elevator 到达指定楼层的时间,elevator 会更新 currentFloor(第 634 ~ 636 行)和 direction(第 638 行和第 639 行)。然后 elevator 调用 arriveAtFloor 工具函数执行到达楼层所需的任务。

processPossibleDeparture 工具函数决定电梯是否需要开始移向另一个需要服务的楼层移动。代码决定当前楼层或其他楼层是否需要 elevator 的服务(第 658 ~ 666 行)。如果当前楼层需要服务,elevator 会对当前楼层调用 arriveAtFloor 函数(第 670 行和第 671 行)。另外,调用 prepareToLeave 工具函数(第 677 行),如果其他楼层需要服务,elevator 会移向该楼层。

ArriveAtFloor 工具函数执行 elevator 到达指定楼层所需完成的任务。该工具函数首先将 moving 成员变量设置为 false(第 683 行)使电梯停止移动,然后重新设置 elevatorButton(第 686 行)并响铃(第 688 行)。接着声明 Person 类对象的临时指针,保存于楼层等待的 Person 对象的句柄。这个指针接收调用楼层的 elevatorArrived 成员函数(第 691 行)的返回值。

Elevator 通过调用 Door 类的 openDoor 成员函数打开门,将一个句柄作为参数传递给当前的乘客,一个句柄传递给在楼层等待的人,一个句柄传递给 elevator 已经到达的楼层,一个句柄传递给 elevator 本身(第 693 行和第 694 行)。Elevator 再次确定楼层是否需要服务(第 696 ~ 704 行)。如果当前楼层不需要 elevator 的服务,elevator 就准备离开向其他楼层(第 709 行),如果其他楼层需要电梯的服务,电梯就会离开。另外,elevator 对当前楼层重新设置服务标记(第 711 行和第 712 行)。

`summonElevator` 成员函数允许其他对象向 `elevator` 要求服务。调用 `summonElevator` 成员函数时,该成员函数取楼层号码作为参数并设置适当的服务标记为 `true`(第 719 行和第 720 行)。

`passengerEnters` 成员函数取 `Person` 类对象的指针作为惟一的参数(第 724 行),并更新 `elevator` 的 `passengerPtr` 句柄使该句柄指向新的乘客(第 727 行)。`passengerExits` 成员函数只是将 `passengerPtr` 句柄设置为 0,从而指示乘客已经离开了 `elevator`(第 735 行)。

`prepareToLeave` 成员函数取一个 `bool` 类型的参数,表明 `elevator` 是否要离开当前楼层(第 738 行)。`Elevator` 通过向楼层发送 `elevatorLeaving` 信息(第 744 行)通知当前楼层 `elevator` 正在离开该楼层。然后 `elevator` 关上 `door`(第 746 行)。最后电梯检查是否要离开楼层(第 748 行),如果是,通过调用 `move` 工具函数(第 749 行)将 `moving` 数据成员设置为 `true`(第 754 行)`elevator` 开始移动。然后通过使用静态常量值 `ELEVATOR_TRAVEL_TIME`(第 757 ~ 758 行)计算 `elevator` 到达目标楼层的到达时间。最后,输出正在移动的 `direction`、目标楼层以及确定的 `arrivalTime`(第 760 ~ 765 行)。

`Floor` 定义(参见图 7.30)中包含将其他类对象与 `Floor` 对象的联系起来的多种方式。首先将 `elevator` 的句柄作为引用使用(第 804 行),因为句柄总是涉及相同的 `elevator`。还将 `Person` 对象的句柄作为指针(第 805 行),每次乘客步行到楼层或离开楼层进入 `elevator` 这个句柄会改变。最后合成对象,包含一个 `public floorButton` 对象(第 800 行)和一个 `private light` 对象(第 806 行)。将 `floorButton` 声明为 `public`,允许 `Person` 类对象直接访问 `floorButton` 对象。<sup>①</sup> `Floor` 类定义包含静态常量数据成员 `FLOOR1` 和 `FLOOR2`(第 798 行和第 799 行)。我们用这些常量代替实际楼层号码,并在实现文件中将这些常量数据成员初始化(第 821 行至第 822 行)。普通常量数据成员必须在构造函数成员初始化列表中初始化。静态常量数据成员这样的特殊例子在文件范围内初始化。

```

767 //floor.h
768 //Definition for class Floor.
769 #ifndef FLOOR_H
770 #define FLOOR_H
771
772 #include "floorButton.h"
773 #include "light.h"
774
775 class Elevator;           //forward declaration
776 class Person;             //forward declaration
777
778 class Floor {
779
780 public:
781     Floor( int, Elevator & );    //constructor
782     ~Floor();                  //destructor
783     bool isOccupied() const;    //return true if floor occupied
784     int getNumber() const;      //return floor's number

```

① 乘客通常不允许接触楼层显示灯(除非该人相当专业),因此 `light` 对象在类定义的 `private` 部分进行声明。

```

785
786 //pass a handle to new person coming on floor
787 void personArrives( Person * const );
788
789 //notify floor that elevator has arrived
790 Person *elevatorArrived();
791
792 //notify floor that elevator is leaving
793 void elevatorLeaving();
794
795 //notify floor that person is leaving floor
796 void personBoardingElevator();
797
798 static const int FLOOR1;
799 static const int FLOOR2;
800 FloorButton floorButton; //floorButton object
801
802 private;
803     const int floorNumber;      //the floor's number
804     Elevator &elevatorRef;      //pointer to elevator
805     Person *occupantPtr;        //pointer to person on floor
806     Light light;                //light object
807 };
808
809 #endif //FLOOR_H

```

图 7.30 Floor 类的头文件

图 7.31 包含 Floor 类的实现文件。Floor 的 isOccupied 成员函数(第 836 ~ 838 行)返回一个 bool 值,表明楼层是还有人在等候。要确定是否有人在楼层等候,应检查 occupantPtr 是否非 0(第 838 行)。如果 occupantPtr 是 0,表明楼层上没有人。getNumber 成员函数返回 floorNumber 成员变量的值(第 841 行)。personArrives 成员函数接到在楼层等候的 Person 对象的指针。该指针赋给 private 数据成员 occupantPtr。

```

810 //floor.cpp
811 //Member function definitions for class Floor.
812 #include <iostream>
813
814 using std::cout;
815 using std::endl;
816
817 //#include <cstring>
818 #include "floor.h"
819 #include "person.h"
820 #include "elevator.h"
821
822 const int Floor::FLOOR1 = 1;
823 const int Floor::FLOOR2 = 2;
824
825 //constructor
826 Floor::Floor(int number, Elevator &elevatorHandle)

```

```

827     ; floorButton( number, elevatorHandle ),
828     floorNumber( number ), elevatorRef( elevatorHandle ),
829     occupantPtr ( 0 ),
830     light( floorNumber == 1 ? "floor 1" : "floor 2" )
831 | cout << "floor " << floorNumber << " created" << endl; |
832
833 Floor::~Floor() //destructor
834 | cout << "floor " << floorNumber << " destroyed" << endl; |
835
836 //determine if floor is occupied
837 bool Floor::isOccupied() const
838 | return ( occupantPtr != 0 ); |
839
840 //return this floor's number
841 int Floor::getNumber() const | return floorNumber; |
842
843 //pass person to floor
844 void Floor::personArrives( Person * const personPtr )
845 | occupantPtr = personPtr; |
846
847 //notify floor that elevator has arrived
848 Person *Floor::elevatorArrived()
849 |
850     //reset the button on floor, if necessary
851     cout << "floor " << floorNumber
852         << " resets its button" << endl;
853     floorButton.resetButton();
854
855     light.turnOn();
856
857     return occupantPtr;
858 |
859
860 //tell floor that elevator is leaving
861 void Floor::elevatorLeaving() | light.turnOff(); |
862
863 //notifies floor that person is leaving
864 void Floor::personBoardingElevator() | occupantPtr = 0; |

```

图 7.31 Floor 类的实现文件

elevatorArrived 成员函数(第 847 ~ 858 行)重新设置 floor 的 floorButton 对象(第 853 行),打开 light 并返回 occupantPtr 句柄(第 857 行)。elevatorLeaving 成员函数关上 light(第 861 行)。最后 personBoardingElevator 成员函数将 occupantPtr 设置为 0,表明乘客已经离开楼层(第 864 行)。

现在我们已经比较熟悉 Person 类头文件中的元素(参见图 7.32),getID 成员函数返回 Person 对象惟一的 ID。setOntoFloor,enterElevator 和 exitElevator 成员函数形成 Person 的 public 接口的其余部分。我们通过 private 静态类变量 personCount 得知如何生成多个 Person 类对象。我们还将 ID 和 destinationFloor 属性声明为 private 常量数据成员。

```

865 //person.h
866 //definition of class Person
867 #ifndef PERSON_H
868 #define PERSON_H
869
870 class Floor;           //forward declaration
871 class Elevator;        //forward declaration
872
873 class Person {
874
875 public:
876     Person( const int );    //constructor
877     ~Person();              //destructor
878     int getID() const;      //returns person's ID
879
880     void stepOntoFloor( Floor & );
881     void enterElevator( Elevator &, Floor & );
882     void exitElevator( const Floor &, Elevator & ) const;
883
884 private:
885     static int personCount; //total number of persons
886     const int ID;           //person's unique ID #
887     const int destinationFloor; //destination floor #
888 };
889
890 #endif //PERSON_H

```

图 7.32 Person 类的头文件

person 类的实现(图 7.33)从构造函数开始(第 905 ~ 907 行),该构造函数取一个常量整数为参数,描绘了 Person 对象目标楼层,模拟系统的输出结果中使用此值。析构函数(第 909 ~ 914 行)显示了一条消息,表明乘客已经离开电梯。

```

891 //person.cpp
892 //Member function definitions for class Person.
893 #include <iostream>
894
895 using std::cout;
896 using std::endl;
897
898 #include "person.h"
899 #include "floor.h"
900 #include "elevator.h"
901
902 //initialize static member personCount
903 int Person::personCount = 0;
904
905 Person::Person( const int destFloor ) //constructor
906     : ID( ++personCount ), destinationFloor( destFloor )
907 {}
908

```

```

909 Person::~~Person() //destructor
910 {
911     cout << "person " << ID << " exits simulation on floor "
912         << destinationFloor << " (person destructor invoked)"
913         << endl;
914 }
915
916 int Person::getID() const { return ID; } //get the ID
917
918 //person walks onto a floor
919 void Person::stepOntoFloor( Floor& floor )
920 {
921     //notify floor a person is coming
922     cout << "person " << ID << " steps onto floor "
923         << floor.getNumber() << endl;
924     floor.personArrives( this );
925
926     //press button on the floor
927     cout << "person " << ID
928         << " presses floor button on floor "
929         << floor.getNumber() << endl;
930     floor.floorButton.pressButton();
931 }
932
933 //person enters elevator
934 void Person::enterElevator( Elevator &elevator, Floor &floor )
935 {
936     floor.personBoardingElevator(); //person leaves floor
937
938     elevator.passengerEnters( this ); //person enters elevator
939
940     //press button on elevator
941     cout << "person " << ID
942         << " presses elevator button" << endl;
943     elevator.elevatorButton.pressButton();
944 }
945
946 //person exits elevator
947 void Person::exitElevator(
948     const Floor &floor, Elevator &elevator ) const
949 {
950     cout << "person " << ID << " exits elevator on floor "
951         << floor.getNumber() << endl;
952     elevator.passengerExits();
953 }

```

图 7.33 Person 类的实现文件

成员函数 `stepOntoFloor` (第 918 ~ 931 行) 通过向楼层发送 `personArrives` 信息 (第 924 行), 通知楼层乘客已经到达楼层, 接着乘客调用 `floorButton` 的 `pressButton` 方法 (第 930 行), 召唤电梯。



enterElevator 成员函数通过发送 personBoardingElevator 信息(第 936 行),首先通知楼层乘客已经进入 elevator。乘客发送 passengerEnters 信息通知 elevator 乘客正在进入电梯(第 938 行),然后乘客向 elevatorButton 对象发送 pressButton 信息,使 elevator 开始移向其他楼层(第 943 行)。exitElevator 成员函数输出一条信息,表明乘客正在离开电梯,然后将 passengerExits 信息发送给 elevator。

我们此时已经完成了第 2 章提到的电梯模拟程序实现方法。第 8 章没有提供“对象思想”小节。第 9 章将介绍 C++ 中的继承以及如何将其应用于电梯模拟系统。

## 7.12 小结

- 不能修改关键字 const 指定的对象。
- C++ 编译器不允许任何非常量成员函数调用常量对象。
- 试图通过类的常量成员函数修改该类对象的数据成员是语法错误。
- 函数在原型和定义中指定为常量。
- 常量成员函数可以用非常量版本重载,编译器根据对象是否声明为常量自动选择所用的重载版本。
- 常量对象必须初始化。要用成员初始化值向构造函数提供类对象数据成员的初始化值。
- 类可以由其他类对象组成。
- 成员对象按声明的顺序在构建所在类对象之前构建。
- 如果不提供成员初始化值,由隐含调用成员对象的默认值构造函数。
- 类的友元函数在类范围之外定义,但有权访问类的所有成员。
- 友元关系声明可以放在类定义中的任何地方。
- this 指针隐式引用对象的非静态数据成员和非静态成员函数。
- 每个非静态成员函数都通过关键字 this 访问自己对象的地址。
- new 操作符自动生成正确长度的对象,调用对象构造函数和返回正确类型的指针。要释放这个对象的空间,需要使用 delete 操作符。
- 对象数组可以用 new 动态分配,语句

```
int *ptr = new int[100];
```

分配 100 个整数的数组并将数组开始位置指定为 ptr。上述整数数组可以用语句

```
delete [] ptr;
```

删除。

- 静态数据成员表示整个类范围的信息(即类的所有内容,而不止是对象)。静态类成员的声明以关键字 static 开始。
- 静态数据成员有类作用域。
- 类的静态成员可以通过类对象访问或者通过类名用作用域分辨符访问(如果成员是 public 类型)
- 如果成员函数不访问非静态类成员,则可以将该成员函数声明为 static。与非静态成

员函数不同的是,静态成员函数没有 this 指针,因为静态数据成员和静态成员函数是独立于类对象而存在的。

- 类通常对类的客户隐藏实现细节,这称为信息隐藏。
- 堆栈是后进先出(LIFO)的数据结构,最后放进堆栈的项目最先从堆栈中取出。
- 描述与其实现细节无关的类的功能称为数据抽象,C++ 类定义所谓的抽象数据类型(ADT)。
- C++ 提高了数据的重要性。C++ 的主要活动是生成自己的新数据类型(即类)和表达这些数据的相互作用。
- 抽象数据类型实际上是在计算机系统中尽量挖掘表示现实世界的概念。
- 抽象数据类型实际上是数据表达和该数据允许的操作。
- C++ 是一种较为精炼的语言,只向程序员建立各种系统的原始功能。该语言将简化编程。
- 队列中的项目按先进先出(FIFO)顺序出队,第一个插入队列的项目第一个离开队列。
- 容器类(也称为集合类)是保存一组对象集合的类,容器类通常提供插入、删除、查找、排序和测试类成员项目等操作。
- 容器类常与迭代对象(或简称迭代器)关联。迭代对象返回集合中的下一个项目(或对集合中的下一个项目进行某种操作)。
- 向客户代码提供代理类,代理类只能访问类的 public 接口,这样就可以让客户代码使用类的服务而不必访问类的实现细节。
- 代理类惟一的 private 成员是隐藏该类对象的 private 数据的指针。
- 类定义只使用另一个类的指针时,另一个类的头文件(通常显示该类的 private 数据)不需要用#include 包含在内。只要在文件中使用该类型之前用正向类声明将另外的类声明为一种数据类型即可。
- 实现文件包含代理类成员函数,该文件包含隐藏该类 private 数据的头文件。
- 实现文件以预处理对象文件形式和头文件一起提供给客户代码,该头文件包含代理类提供服务的函数原型。

## 本章术语

abstract data type (ADT) 抽象数据类型

binary scope resolution operator (::)

二元作用域分辨符(::)

cascading member function calls

连续使用成员函数调用

class scope 类作用域

composition 合成

const member function 常量成员函数

const object 常量对象

constructor 构造函数

container 容器

data representations 数据表达

default constructor 默认构造函数

default destructor 默认析构函数

delete operator delete 操作符

delete[] operator delete[] 操作符

dequeue operation 出队(队列操作)

destructor 析构函数

dynamic objects 动态对象

enqueue(queue operation) 入队(队列操作)

extensible language 可扩展语言

first-in-first-out(FIFO) 先进先出

|                                |              |                                          |                 |
|--------------------------------|--------------|------------------------------------------|-----------------|
| forward class declaration      | 正向类声明        | object-based programming                 | 基于对象编程          |
| friend class                   | 友元类          | operations in an ADT                     | ADT 中的操作        |
| friend function                | 友元函数         | pointer member selection operation( -> ) | 指针成员选择操作符( -> ) |
| host object                    | 宿主对象         | pop( stack operation )                   | 弹出( 堆栈操作 )      |
| iterator                       | 迭代器          | principle of least privilege             | 最低权限原则          |
| last-in-first-out( LIFO )      | 后进先出         | proxy class                              | 代理类             |
| member access specifiers       | 成员访问说明符      | push( stack operation )                  | 压入( 堆栈操作 )      |
| member initializer             | 成员初始化值       | queue abstract data type                 | 队列抽象数据类型        |
| member object                  | 成员对象         | stack abstract data type                 | 堆栈抽象数据类型        |
| member object constructor      | 成员对象构造函数     | static data member                       | 静态数据成员          |
| member selection operator( . ) | 成员选择操作符( . ) | static member function                   | 静态成员函数          |
| new operator                   | new 操作符      | this pointer                             | this 指针         |
| new[ ] operator                | new[ ] 操作符   |                                          |                 |

## “对象思想”术语

|                          |        |                     |      |
|--------------------------|--------|---------------------|------|
| circular include problem | 循环包容问题 | forward declaration | 提前声明 |
|--------------------------|--------|---------------------|------|

## 常见编程错误

- 7.1 把修改对象数据成员的成员函数定义为 const 是语法错误。
- 7.2 将调用同一类实例的非常量成员函数定义为 const 是语法错误。
- 7.3 对常量对象调用非常量成员函数是语法错误。
- 7.4 将构造函数和析构函数声明为 const 是语法错误。
- 7.5 不为常量数据成员提供成员初始化值是语法错误。
- 7.6 既不为成员对象提供成员初始化值,又不为成员对象的类提供默认的构造函数会造成语法错误。
- 7.7 打算同时使用对象指针和成员选择操作符( . )是语法错误,因为成员选择操作符和对象或该对象的引用一起使用。
- 7.8 将 new 和 delete 类型的动态分配内存方法与 malloc 和 free 类型的动态分配内存方法混用是逻辑错误;malloc 分配的空间无法用 delete 释放,new 分配的空间无法用 free 删除。
- 7.9 删除数组时用 delete 代替 delete[ ] 将导致运行时的逻辑错误。为避免这一错误,数组生成的内存空间要用 delete[ ] 操作符删除,各个元素生成的内存空间则用 delete 操作符删除。
- 7.10 文件范围内静态类变量定义中使用关键字 static 是语法错误。
- 7.11 在静态成员函数中引用 this 指针是语法错误。
- 7.12 将静态成员函数声明为 const 是语法错误。

## 良好编程习惯

- 7.1 将所有无须修改当前对象的成员函数声明为 const,以便在需要时调用常量对象。
- 7.2 将类中所有友元关系声明放在类的首部之后,不要在其前面添加任何成员访问说明符。
- 7.3 因为 C++ 语言包含 C 语言,所以 C++ 程序可以同对包含用 malloc 生成和用 free 删除的

存储空间以及用 new 生成和用 delete 删除的对象。但最好尽量使用 new 和 delete。

- 7.4 删除动态分配内存后,将指向该内存的指针设置为指向 0,以断开指针与前面已分配内存的连接。

### 性能提示

- 7.1 变量和对象声明为 const 不仅是有效的软件工程原则,还能提高性能,因为如今复杂的优化编译器能对常量进行某些无法对变量进行的优化。
- 7.2 通过成员初始化值显式初始化成员对象,可以避免两次初始化成员对象的开销:一次是调用成员对象的默认构造函数;另一次是用 set 函数初始化成员对象。
- 7.3 为了节约内存空间,每个类的每个成员都只有一个副本,该类的每个对象都可调用这个成员函数。另一方面,每个对象又有自己的类数据成员副本。
- 7.4 一个数据副本够用时,用静态数据成员则可以节省存储空间。

### 软件工程知识

- 7.1 将对象声明为 const 有利于实现最低权限原则,试图修改对象会产生编译时错误而非运行时错误。
- 7.2 对于正确的类设计、程序设计和编码使用 const 是非常关键的。
- 7.3 常量成员函数可以用非常量版本重载。编译器根据对象是否为常量对象自动选择所用的重载版本。
- 7.4 常量对象不能用赋值语句修改,所以必须初始化。当类的数据成员声明为常量时,要用成员初始化值向构造函数提供类对象数据成员的初始值。
- 7.5 常量类成员(常量对象和常量变量)要用成员初始化值语法初始化,不能用赋值语句。
- 7.6 如果成员函数不修改对象,最好将所有的类成员函数声明为常量。如果不需要生成该类的常量类型对象,就没有必要这样做。将这种成员函数声明为常量有一个好处,如果无意中修改了这个成员函数中的对象,编译器会产生语法错误消息。
- 7.7 合成是软件重用的最常见形式之一,即一个类可以将其他类的对象作为自己的成员。
- 7.8 如果一个类将其他类的对象作为其成员,即使将该成员对象指定为 public,也不会破坏该成员对象 private 成员的封装与隐藏。
- 7.9 尽管类定义中有友元函数的原型,但友元仍然不是成员函数。
- 7.10 private,protected 和 public 的成员访问符号与友元关系的声明无关,因此友元关系声明可以放入类定义中的任何位置。
- 7.11 OOP 组织中,有人认为友元关系破坏了信息隐藏并削弱了面向对象设计方法的优势。
- 7.12 由于 C++ 属于混合语言,常在程序中同时采用两种函数调用且两种函数调用往往相反,那么 C 语言的调用将基本数据或对象传递给函数,C++ 调用则是将函数(或信息)传递给对象。
- 7.13 有些公司的软件工程标准中,明确规定所有静态成员函数只能调用类名句柄,不能调用对象句柄。
- 7.14 即使类对象尚未初始化,类的静态数据成员和成员函数也可以存在并使用。

- 7.15 程序员可以通过类机制生成新类型。这些新类型设计好之后,可以像使用内部类型一样方便地使用,所以我们说C++是可扩展的语言。尽管该语言可以随新类型扩展,但是基础语言本身是不会改变的。

### 测试和调试提示

- 7.1 如果成员函数不修改对象,那就始终将其声明为常量,以尽量避免错误。
- 7.2 类似C++的语言是不断变化的,新的关键字会不断出现。不要用“object”之类的标识符。尽管“object”目前还不是C++中的关键字,但将来可能会变成关键字,新的编译器可能不能接受现有的代码。

### 自测题

#### 7.1 填空题:

- a) \_\_\_\_\_语法用于初始化类的常量成员。
- b) 成员函数应声明为类的\_\_\_\_\_才能访问类的 private 数据成员。
- c) \_\_\_\_\_操作符对指定类型对象动态分配内存并返回该类型的\_\_\_\_\_。
- d) 常量对象应\_\_\_\_\_,不能在生成之后修改。
- e) \_\_\_\_\_数据成员表示类范围信息。
- f) 对象的成员函数能访问对象的“自我指针”,称为\_\_\_\_\_指针。
- g) 关键字\_\_\_\_\_指定对象或变量初始化之后不可修改。
- h) 如果类的成员对象不提供成员初始化值,则调用该对象的\_\_\_\_\_。
- i) 如果成员函数不访问\_\_\_\_\_类成员,则可以声明为 static。
- j) 成员对象在所在类对象之\_\_\_\_\_构建。
- k) \_\_\_\_\_操作符删除前面用 new 分配的内存。

#### 7.2 指出下列各题中的错误,并说明如何改正:

- a) 

```
class Example {
public:
    Example (int y=10) {data=y;}
    int getIncrementedData () const {return ++data;}
    static int getCount ()
    {
        cout << "Data is" << data << endl;
        return count;
    }
private:
    int data;
    static int count;
}
```
- b) 

```
char *string;
string = new char [20];
free (string);
```

## 自测题答案:

- 7.1 a) 成员初始化值    b) 友元    c) new, 指针    d) 初始化    e) 静态  
 f) this    g) const    h) 默认构造函数    i) 非静态    j) 前    k) delete
- 7.2 a) 错误: Example 的类定义有两个错误。第一个在 getIncrementedData 函数中, 函数声明为 const, 却修改了对象。  
 改正: 要改正第一个错误, 删除 getIncrementedData 定义中的 const 关键字即可。  
 错误: 第二个错误在 getCount 函数中。函数声明为 static, 所以不能访问类的非静态成员。  
 改正: 要改正第二个错误, 删除 getCount 定义中的输出行即可。
- b) 错误: 用 C 标准库函数 free 删除 new 动态分配的内存。  
 改正: 用 C++ 的 delete 操作符释放内存。C 语言类型的动态内存分配操作符不能与 C++ 的 new 和 delete 操作符混用。

## 练习题

- 7.3 比较用于动态内存分配的 C++ 的 new 和 delete 操作符与 C 标准库函数 malloc 和 free 操作符。
- 7.4 说说 C++ 中友元关系的概念及其副作用。
- 7.5 正确的 Time 类定义能否同时包括下列构造函数? 如果不能, 请说明原因。

```
Time( int h=0, int m=0, int s=0);
Time()
```

- 7.6 构造函数和析构函数指定返回类型(即使 void)时, 会出现什么情况?
- 7.7 用下列条件生成 Date 类:

- a) 用多种格式输出日期

```
DDD YYYY
MM/DD/YY
June 14, 1992
```

- b) 用重载的构造函数生成 Date 对象, 用 a) 中的日期格式进行初始化。
- c) 创建一个 Date 构造函数, 用 time.h 头文件的标准库函数读取系统日期和设置 Date 成员。

第8章将创建操作符, 用于测试两个日期的相等性和比较两个日期的先后顺序。

- 7.8 创建一个 SavingsAccount 类。用静态数据成员包含每个存款人的 annualInterestRate(年利率)。类的每个成员包含一个 private 数据成员 savingsBalance, 表示当前存款额。提供一个 calculateMonthlyInterest 成员函数, 计算月利息, 用 balance 乘以 annualInterestRate 除以 12 取得, 并将当月月息加入 savingsBalance。提供一个静态成员函数 modifyInterestRate, 将静态 annualInterestRate 设置为新值。实例化两个不同的 SavingsAccount 对象 saver1 和 saver2, 余额分别为 2 000.00 和 3 000.00。将 annualInterestRate 设置为 3%, 计算每个存款人的月息并打印新的结果。

7.9 生成一个 `IntegerSet` 类。`IntegerSet` 类的每个对象可以保存到 0 到 100 之间的整数值。一个集合内部表示为 0 和 1 的数组。数组元素 `a[i]` 为 1 表示整数 `i` 在集合中,数组元素 `a[j]` 为 0 表示整数 `j` 不在集合中。默认构造函数将集合初始化为“空集”,即所有元素都是 0。

提供常用集合操作的成员函数。例如,提供一个 `unionOfIntegerSets` 成员函数,生成两个现有集合的并集(即其中一个集合的元素为 1,则并集的元素就是 1,如果两个集合的元素均为 0,则并集的元素就是 0)。

提供 `intersectionOfIntegerSets` 成员函数生成两个现有集合的交集(即只要其中一个集合的元素为 0,交集的元素就为 0,如果两个集合的元素均为 1,交集的元素就为 1)。

提供一个 `insertElement` 成员函数,在集合中插入新整数 `k`(将 `a[k]` 设置为 1)。提供 `deleteElement` 删除整数 `m`(将 `a[m]` 设置为 0)。

提供一个 `setPrint` 成员函数,将集合表示的值打印为以空格分隔的列表。只打印集合中存在的元素(即对应值为 1 的位置),对于空集,则打印 - - -。

提供一个 `isEqualTo` 成员函数,判断两个集合是否相等。

提供另一个构造函数,取 5 个整数参数,用以初始化一组对象。如果提供的元素少于 5 个,其他元素就用默认参数 -1。

编写一个驱动程序,测试 `IntegerSet` 类。实例化多个 `IntegerSet` 对象,测试所有成员函数能否正确工作。

7.10 图 7.8 中的 `Time` 类可以在内部将时间表示为从午夜算起的秒数而不是 3 个整数值 `hour`, `minute` 和 `second`。客户代码可以用相同的 `public` 方法并取得相同结果。修改图 7.8 中的 `Time` 类,将时间表示为从午夜算起的秒数,证明类的客户代码看不出功能上的变化。

# 第 8 章 操作符重载

## 学习目标

- 了解如何重新定义(重载)操作符以处理新类型
- 了解如何将一个类的对象转换为另一个类的对象
- 了解何时可用,何时不可用重载操作符
- 了解几个有趣的操作符重载示例
- 能创建数组、字符串和数据类

## 8.1 简介

第 6 章和第 7 章介绍了 C++ 类的基本知识和抽象数据类型的概念。对类对象(即抽象数据类型的实例)的操作是通过向对象发送信息来完成的(以成员函数调用的形式)。对于某些类(特别是数学类)来说,这种调用符号较为复杂,但用 C++ 丰富的内部操作符集来指定对象操作非常可取。本章将介绍如何结合使用 C++ 中的操作符和类对象,这个过程称为操作符重载。扩展 C++ 语言使其具有这些新功能是理所当然的。但是还需要注意,因为错误使用重载会使程序难以理解。

操作符 << 在 C++ 中用途较广,既可以用作流插入操作符又可以用作左移位操作符,这是操作符重载的典型示例。同样地,操作符 >> 也是 C++ 中的重载操作符,它既可以用作流读取操作符,也可用作右移位操作符。注意,第 16 章将详细介绍左移位操作符和右移位操作符。C++ 类库中这两个操作符都可以重载。C++ 本身也重载了 + 和 -, 这两个操作符在整数算术运算、浮点数算术运算和指针算术运算等场景中执行的运算是不同的。

C++ 允许程序员重载大多数操作符,使其更符合使用场景。编译器根据操作符的使用方式生成合适的代码。一些操作符经常需要重载,特别是赋值操作符和各种数学操作符,如 + 和 -。虽然重载操作符实现的任务也可以通过显式函数调用来完成,但是使用重载操作符可以使程序更清晰。

我们将介绍何时可用操作符重载,何时又不可用。此外还将演示如何使用操作符,并介绍使用重载操作符的完整程序。

## 8.2 操作符重载的基础知识

C++ 程序设计对类型敏感,并且程序设计的重点在于类型。程序员可以使用内部类型,也可以定义新类型。内部类型可以与 C++ 中丰富的操作符集结合使用。操作符为程序员提供了表示内部类型对象操作的简略表达式符号。



程序员也可以把操作符和用户自定义类型结合使用。尽管C++不允许生成新的操作符,但是允许重载现有的大多数操作符,使这些操作符可以与类对象结合使用,操作符具有新类型的含义,这是C++语言最强大的功能之一。

**软件工程知识 8.1** 操作符重载提高了C++的可扩展性,这也是C++最吸引人的属性之一。

**良好编程习惯 8.1** 针对同样的操作,使用重载操作符比使用显式函数调用更能提高程序的可读性。

**良好编程习惯 8.2** 由于过多或前后不一致地使用操作符重载会使程序晦涩难懂,所以应尽量避免。

虽然操作符重载似乎是C++的外部功能,但大多数程序员经常都隐式使用重载操作符。例如,加法操作符(+)对整数、浮点数和复数的操作是很不同的。但是,因为C++本身已经重载了加法操作符(+),所以该操作符能够很好地用于int, float, double类型的变量和其他内部类型的变量。

操作符重载是通过编写函数定义(包函数首部和函数体)来实现的,除非被重载的操作符号前的函数名变成关键字operator。例如,函数名operator+用于重载操作符(+)

用于类对象的操作符必须重载,但有两种情况例外。赋值操作符(=)无须显式重载就可以用于每一个类。赋值操作符的默认行为是类数据成员的成员赋值。这种默认赋值行为对于带有指针成员类是危险的,我们将对这种类显示重载赋值操作符。地址操作符(&)同样无须重载即可用于任何类对象,它只返回内存中对象的地址。地址操作符也可以重载。

操作符重载最适合用于数学类。为保证与在现实世界中操作这些数字类的方式一致,通常要重载一组操作符。例如:对于复杂的数字类,通常不仅仅要重载加法操作符(+),因为其他数学操作符也经常用于复杂数字。

C++的操作符很丰富。因为C++程序员理解每个操作符的含义和操作符所在场景,所以在重载用于新的类操作符时,程序员能做出合理选择。

C++为其内部类型提供了丰富的操作符集,重载这些操作符的目的是为用户自定义的类型提供同样简洁的表达式。但操作符的重载不是自动完成的,程序员必须为所要执行的操作编写操作符重载函数,对于这些函数,有时最好用作成员函数,有时最好用作友元函数。在极少数情况下,它们可能既不是成员函数,也不是友元函数。

重载误用的情况可能会发生,例如重载加法操作符(+)使其执行类似减法的运算,或者重载除法操作符(/)使其执行类似乘法的运算。如此误用重载会使程序非常难以理解。

**良好编程习惯 8.3** 重载操作符用于类的对象时,其功能类似于该操作符作用于内部类型的对象时完成的功能,避免无目的地滥用重载操作符。

**良好编程习惯 8.4** 用重载操作符编写C++程序之前,查阅编辑器手册,了解特定操作符的各种限制和要求。

## 8.3 操作符重载的限制条件

C++中的大部分操作符都可以重载。图8.1列举了可以重载的操作符。图8.2列举了

不能重载的操作符。

**常见编程错误 8.1** 试图重载不能重载的操作符是语法错误。

| 可以重载的操作符 |          |    |    |    |    |     |        |
|----------|----------|----|----|----|----|-----|--------|
| +        | -        | *  | /  | %  | ^  | &   |        |
| ~        | !        | =  | <  | >  | += | --  | *=     |
| /=       | %=       | ^= | &= | =  | << | >>  | >>=    |
| <<=      | =        | != | <= | >= | && |     | ++     |
| --       | -> *     | '  | -> | [] | () | new | delete |
| new[]    | delete[] |    |    |    |    |     |        |

图 8.1 可以重载的操作符

| 不能重载的操作符 |   |    |    |        |
|----------|---|----|----|--------|
| .        | * | :: | ?: | sizeof |

图 8.2 不能重载的操作符

重载不能改变操作符的优先级,但这样会导致非常难以处理的局面,因为以其固定优先级方式重载操作符是不恰当的。但是在表达式中使用圆括号可以强制改变重载操作符的计算顺序。

重载不能改变操作符的结合性。

重载不能改变操作符操作数的个数(操作符包含操作数的个数)。一元操作符仍然作为一元操作符重载,二元操作符仍然作为二元操作符重载,C++中唯一的三元操作符(?:)不能重载。操作符 &, \*, + 和 - 既可以用作一元操作符,也可以用作二元操作符,可以分别把它们重载为一元操作符和二元操作符。

不能创建新的操作符,只有现有的操作符才可以重载。这限制了程序员使用一些流行的符号,如 BASIC 和 FORTRAN 中表示指数的操作符 \* \*。

**常见编程错误 8.2** 试图创建新的操作符是语法错误。

操作符重载不能改变该操作符用于内部类型对象时的作用方式。例如,程序员不能改变操作符 + 用于两个整数时的含义。操作符重载只能随用户自定义类型的对象一起使用,或者随用户自定义类型的对象和内部类型的对象混合使用时使用。

**常见编程错误 8.3** 试图改变操作符对内部类型的对象的作用方式是语法错误。

**软件工程知识 8.2** 操作符函数的参数至少有一个必须是类对象或对类对象的引用。这样可防止程序员改变操作符对内部类型的对象的作用方式。

重载了赋值操作符 = 和加法操作符 + 以后,语句

```
object2 = object2 + object1;
```

成立,并不意味着自动重载操作符 +=。语句

```
object2 += object1;
```

也是成立的,但显式重载操作符 += 语句可使上述语句成立。

**常见编程错误 8.4** 误以为重载了某个操作符(如“+”)可以自动重载相关的操作符(如“+=”),重载了“=”就自动重载了“!=”。操作符只能显式重载(不会隐式重载)。

**常见编程错误 8.5** 试图通过操作符重载改变操作符的“个数”是语法错误。

**良好编程习惯 8.5** 要保证相关操作符的一致性,可以用一个操作符实现另一个操作符(例如,用重载的操作符“+”实现重载的操作符“+=”)。

## 8.4 类成员操作符函数与友元函数操作符函数的对比

操作符函数既可以是成员函数,也可以是非成员函数。非成员函数经常因为某种原因被指定为友元函数。成员函数是用 `this` 指针隐式访问类对象的某个参数(二元操作符的左参数),非成员函数的调用必须显式列出该参数。

在重载操作符(),[],->或者任何赋值操作符时,操作符重载函数必须声明为类的一个成员。对于其他操作符,操作符重载函数可以是非成员函数。

不管操作符函数是成员函数还是非成员函数,操作符在表达式中的用法仍然是相同的。哪种实现方式最好呢?

当操作符函数作为成品函数实现时,最左边的操作数(或者只有最左边的操作数)必须是操作符类的一个类对象(或者是对该类对象的引用)。如果左边的操作数是不同类的对象或是一个内部类型的对象,该操作符函数必须作为一个非成员函数来实现(正如分别重载操作符<<和>>作为流插入操作符和流读取操作符一样)。操作符函数作为非成员函数直接访问该类的 `private` 或者 `protected` 成员时,该函数必须是一个友元。

重载<<操作符必须有一个类型为 `ostream &` 的左操作数(如表达式 `cout << classObject` 中的 `cout`),因此它必须是一个非成员函数。类似地,重载>>操作符必须有一个类型为 `istream &` 的左操作数(如表达式 `cin >> classObject` 中的 `cin`),所以它也必须是一个非成员函数。此外,这两个重载的操作符函数都需要访问输出或输入的类对象的 `private` 数据成员,因此出于性能因素考虑,这些重载的操作符函数经常被指定为类的友元函数。

**性能提示 8.1** 可以把一个操作符作为一个非成员、非友元函数重载。但是,这样的操作符函数访问类的 `private` 和 `protected` 数据时必须使用类的 `public` 接口中提供的 `set` 或 `get` 函数(即设置数据和读取数据的函数),调用这些函数所涉及的开销会降低性能,因此必须将这些函数内联以提高性能。

只有当二元操作符最左边的操作数是该类的一个对象时,或者当一元操作符的操作数是该类的一个对象时,才有必要调用特定类的操作符成员函数。

选择非成员函数重载操作符的另一个原因是令操作符具有可交换性。例如:假设我们有一个 `long int` 类型的对象 `number` 和 `HugeInteger1` 类的一个对象 `bigInteger1`(本章练习题开发了 `HugeInteger` 类,该类中的整数可以是任意大小,不受机器字长的限制)。加法操作符(+)生成一个临时的 `HugeInteger` 对象,将其作为 `HugeInteger` 和 `long int` 类型对象之和(如表达式 `bigInteger + number` 所示),或作为 `long int` 和 `HugeInteger` 类型对象的和(如表达式 `number + bigInteger1` 所示),那就说明上述加法操作符须具有可交换性(如同普通的加法)。

问题在于,如果把操作符作为成员函数重载,类对象必须出现在操作符的左边,所以要将操作符函数作为一个非成员的友元函数重载,这样才能允许 HugelInteger 对象出现在加法操作符的右边。处理 HugelInteger 对象在左边的 operator + 函数依然可以是一个成员函数。记住,非成员函数不一定要是友元函数,只要类的 public 接口中有相应 set 和 get 函数, set 和 get 函数若能内联则更好。

## 8.5 重载流插入与流读取操作符

C++ 的流读取操作符 >> 和流插入操作符 << 可用于输入/输出标准类型的数据。C++ 编译器在类库中提供的这两个操作符被重载,处理包括类似 C 语言中的 char \* 字符串和指针在内的各种内部数据类型。流插入和流读取操作符也可以重载用于输入/输出用户自定义类型的数据。图 8.3 中的程序演示了重载的流读取操作符和流插入操作符,用于处理用户自定义的称为 PhoneNumber 类的电话号码的数据。程序假定输入的电话号码是正确的,错误检测留在稍后的练习题中完成。

```

1 //Fig. 8.3: fig08_03.cpp
2 //Overloading the stream-insertion and
3 //stream-extraction operators.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9 using std::ostream;
10 using std::istream;
11
12 #include <iomanip>
13
14 using std::setw;
15
16 class PhoneNumber {
17     friend ostream &operator <<( ostream&, const PhoneNumber & );
18     friend istream &operator >>( istream&, PhoneNumber & );
19
20 private:
21     char areaCode[ 4 ]; //3-digit area code and null
22     char exchange[ 4 ]; //3-digit exchange and null
23     char line[ 5 ]; //4-digit line and null
24 };
25
26 //Overloaded stream-insertion operator (cannot be
27 //a member function if we would like to invoke it with
28 //cout << somePhoneNumber;).
29 ostream &operator <<( ostream &output, const PhoneNumber &num )
30 |
31     output << "(" << num.areaCode << " ) "
32         << num.exchange << " - " << num.line;
```

```

33 return output; //enables cout << a << b << c;
34 }
35
36 istream &operator >>( istream &input, PhoneNumber &num )
37 {
38     input.ignore();                //skip (
39     input >> setw( 4 ) >> num.areaCode; //input area code
40     input.ignore( 2 );              //skip ) and space
41     input >> setw( 4 ) >> num.exchange; //input exchange
42     input.ignore();                //skip dash ( - )
43     input >> setw( 5 ) >> num.line;  //input line
44     return input;                  //enables cin >> a >> b >> c;
45 }
46
47 int main()
48 {
49     PhoneNumber phone;              //create object phone
50
51     cout << "Enter phone number in the form (123) 456 -7890;\n";
52
53     //cin >> phone invokes operator >> function by
54     //issuing the call operator >>( cin, phone ).
55     cin >> phone;
56
57     //cout << phone invokes operator << function by
58     //issuing the call operator <<( cout, phone ).
59     cout << "The phone number entered was; " << phone << endl;
60     return 0;
61 }

```

输出结果:

```

Enter phone number in the form (123) 456 7890;
(800) 555 -1212
The phone number entered was; (800) 555 -1212

```

图 8.3 用户自定义的流插入操作符和流读取操作符

流读取操作符函数 `operator >>` (第 36 行) 含有两个参数, 一个是对 `istream` 的引用, 名为 `input`, 另一个是对用户自定义类型 `PhoneNumber` 的引用, 名为 `num`, 并且函数返回一个 `istream` 引用。操作符函数 `operator >>` 用于将下述格式的电话号码

(800) 555 -1212

输入 `PhoneNumber` 类的对象。编译器遇到 `main()` 函数中的表达式

`cin >> phone`

时, 编译器将生成函数调用

`operator >> (cin, phone);`

执行该调用时, 引用参数 `input` 成为 `cin` 的别名, `num` 成为 `phone` 的别名。操作符函数使用 `istream` 的成员函数 `getline`, 将电话号码的 3 部分作为字符串分别读入被引用的 `PhoneNumber` 对象 (操作符函数中的 `num` 和 `main` 函数中的 `phone`) 的 `areaCode`、`exchange` 和 `line` 成员。流

操纵元 `setw` 保证将正确的字符数读入字符数组。我们曾经用 `cin` 和 `setw` 限制读入的字符数比参数少 1 (例如, `setw(4)` 只允许读入 3 个字符, 留出一个位置保存空中止符)。通过调用 `istream` 的成员函数 `ignore` 跳过括号、空格、破折号等等 (`ignore` 函数删除输入流中指定数目的字符, 默认个数为 1)。函数 `operator >>` 返回对 `istream` 对象的引用 `input` (即 `cin`), 所以能在 `PhoneNumber` 对象的输入操作完成之后, 继续执行对 `PhoneNumber` 的其他对象或者其他数据类型对象的输入操作。例如, 可以像下面这样输入两个 `PhoneNumber` 对象

```
cin >> phone1 >> phone2;
```

首先, 表达式 `cin >> phone1` 将通过调用

```
operator >> (cin, phone1);
```

得以执行。该调用返回 `cin` 引用, 并将其作为 `cin >> phone1` 的值, 因此表达式的其余部分将被解释为 `cin >> phone2`, 这将通过下列调用

```
operator >> (cin, phone2);
```

得以执行。流插入操作符有两个参数, 一个是对 `ostream` 的引用 (即 `output`), 另一个是对用户自定义类型 `PhoneNumber` 的引用 (即 `num`), 并且返回一个对 `ostream` 的引用。函数 `operator <<` 显示了 `PhoneNumber` 的对象。编译器遇到 `main` 函数中的表达式

```
cout << phone
```

时, 将生成非成员函数调用

```
operator << (cout, phone);
```

因为电话号码各部分是以字符串格式保存的, 所以函数 `operator <<` 以字符串形式显示它们。

注意, 函数 `operator <<` 和 `operator >>` 在 `PhoneNumber` 类中被声明为非成员友元函数。由于要把 `PhoneNumber` 类对象作为操作符的右操作数, 所以这些操作符函数必须是非成员函数; 类操作数必须出现在操作符的左边将操作符作为成员函数重载。考虑到性能因素, 如果输入/输出操作符要直接访问非 `public` 类成员, 就要把输入/输出操作符声明为 `friend`。另外, 还要注意 `operator <<` 参数表中的 `PhoneNumber` 引用是常量类型 (因为只输出 `PhoneNumber`), 而 `operator >>` 参数表中的 `PhoneNumber` 引用是非常量类型 (因为 `PhoneNumber` 对象必须修改为在该对象中保存输入的电话号码)。

**软件工程知识 8.3** 不修改类 `ostream` 和 `istream` 的声明和 `private` 数据成员, 也可为用户自定义类型添加新的输入/输出功能。这又一次证明了 C++ 编程语言的可扩展性。

## 8.6 重载一元操作符

类的一元操作符可重载为无参数的非静态成员函数或者带有一个参数的非成员函数, 参数必须是用户自定义类型的对象或者对对象的引用。实现重载操作符的成员函数应是非静态, 以便访问类的非静态数据。记住, 静态成员函数只能访问类的静态数据成员。

本章稍后要用重载的一元操作符 (!) 测试 `String` 类对象是否为空并返回布尔值。把一元操作符 (如 “!”) 重载为无参数的非静态成员函数时, 如果 `s` 是 `String` 类对象或是 `String` 类对象的引用, 那么编译器在遇到表达式 `!s` 时, 会生成函数调用 `s.operator !()`。操作数 `s` 是类的对象, 它调用了 `String` 类的成员函数 `operator !`。类定义中的函数声明

```
class String{
public:
    bool operator! () const;
    ...
};
```

把一元操作符(如“!”)重载为带有一个参数的非成员函数时,参数有两种情况:一是该参数是某个对象(需要对象的副本,以便函数不会对原始对象产生负面影响),二是该参数是某个对象的引用(不复制原始对象,所以函数的所有负面影响都会作用于原始对象)。如果 *s* 是 *String* 类的一个对象或(或是 *String* 类对象的引用),*!s* 就会被处理为 *operator!(s)*,调用 *String* 类的非成员友元函数。*String* 类声明如下

```
class String {
    friend bool operator! (const String &);
    ...
};
```

**良好编程习惯 8.6** 重载一元操作符时,把操作符函数用作类的成员而非友元函数。因为友元函数的使用破坏了类的封装,所以除非绝对必要,否则应尽量避免使用友元函数和友元类。

## 8.7 重载二元操作符

二元操作符可以重载为带有一个参数的非 *static* 成员函数,或者带有两个参数的非成员函数(参数之一必须是类的对象或者是类对象的引用)。

本章稍后要重载操作符 *+=* 以结合 2 个字串对象。把它重载为带有一个参数的 *String* 类的非静态成员函数时,如果 *y* 和 *z* 是 *String* 类的对象,*y += z* 就会被处理为 *y.operator += (z)*,调用具有声明

```
class String{
public:
    const String &operator +=(const String &);
    ...
};
```

的成员函数 *operator +=*。

二元操作符 *+=* 也可以重载为带有两个参数的非成员函数,其中的一个参数必须是类的对象或类对象的引用。如果 *y* 和 *z* 是 *String* 类的对象,*y += z* 就会被处理为 *operator += (y,z)*,调用具有声明

```
class String{
    friend const String &operator +=(String &,
                                    const String &);
    ...
};
```

的友元函数 *operator +=*。

## 8.8 案例分析: Array 类

C++ 中,数组(Array)是指针的替代物,所以数组非常容易出错。例如,程序很可能由于 C++ 不检测下标是否超出数组边界而出现越界错误;大小为  $n$  的数组的下标必须是  $0, 1, 2, \dots, n-1$ , 下标范围是不允许改变的;不能一次输入或输出整个非 char 数组,只能单独读取或者输出每个数组元素;不能用相等操作符或者关系操作符比较两个数组(因为数组名仅仅是指向内存中数组起始位置的指针);当把一个数组传递给一个能处理任意大小数组的常用函数时,数组的大小也必须作为一个额外的参数传给该函数;不能用赋值操作符把一个数组赋给另一个数组(因为数组名是常量指针,常量指针是不能用于赋值操作符的左边的)。尽管所有这些处理能力似乎理所当然, C++ 却没有提供这种能力。不过它提供了实现这种能力的手段,那就是操作符重载。

在这个案例中我们开发了一个 Array 类,它能检测范围以确保数组下标不会越界,允许用赋值操作符把一个数组赋给另一个数组。数组类对象自动知道数组的大小,因而不需将数组的大小传送给函数。可以用流读取操作符和流插入操作符输入/输出整个数组。还可以用相等操作符 `==` 和 `!=` 比较数组。示例程序中的数组类用一个静态成员了解程序中实例化数组对象的个数。

本案例将充分体现数据抽象的概念。当然,还可以增加数组类的其他功能,类的开发十分有趣并富有挑战性,总是把“生成有价值的类”作为目标。

图 8.4 中的程序演示了 Array 类和用于该类的重载操作符。首先看看 main 函数中的驱动程序,然后再探讨类定义以及类的每个成员和友元函数的定义。

```

1 //Fig. 8.4: array1.h
2 //Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream>
7
8 using std::ostream;
9 using std::istream;
10
11 class Array {
12     friend ostream &operator <<( ostream &, const Array & );
13     friend istream &operator >>( istream &, Array & );
14 public:
15     Array( int = 10 );           //default constructor
16     Array( const Array & );      //copy constructor
17     ~Array();                   //destructor
18     int getSize() const;         //return size
19     const Array &operator =( const Array & ); //assign arrays
20     bool operator ==( const Array & ) const; //compare equal
21
22     //Determine if two arrays are not equal and

```



```

23 //return true, otherwise return false (uses operator ==).
24 bool operator! =( const Array &right ) const
25     { return !( *this == right ); }
26
27 int &operator[] ( int );           //subscript operator
28 const int &operator[] ( int ) const; //subscript operator
29 static int getArrayCount();       //Return count of
30                                   //arrays instantiated.
31 private:
32     int size; //size of the array
33     int *ptr; //pointer to first element of array
34     static int arrayCount; // # of Arrays instantiated
35 };
36
37 #endif

```

图 8.4 用操作符重载 Array 类——array1.h

```

38 //Fig 8.4; array1.cpp
39 //Member function definitions for class Array
40 #include <iostream>
41
42 using std::cout;
43 using std::cin;
44 using std::endl;
45
46 #include <iomanip>
47
48 using std::setw;
49
50 #include <cstdlib>
51 #include <cassert>
52 #include "array1.h"
53
54 //Initialize static data member at file scope
55 int Array::arrayCount = 0; //no objects yet
56
57 //Default constructor for class Array (default size 10)
58 Array::Array( int arraySize )
59 {
60     size = ( arraySize > 0 ? arraySize : 10 );
61     ptr = new int[ size ];           //create space for array
62     assert( ptr != 0 );              //terminate if memory not allocated
63     ++arrayCount;                   //count one more object
64
65     for ( int i = 0; i < size; i ++ )
66         ptr[ i ] = 0;               //initialize array
67 }
68
69 //Copy constructor for class Array
70 //must receive a reference to prevent infinite recursion
71 Array::Array( const Array &init ) : size( init.size )

```

```

72 |
73   ptr = new int[ size ];           //create space for array
74   assert( ptr != 0 );              //terminate if memory not allocated
75   ++arrayCount;                   //count one more object
76
77   for ( int i = 0; i < size; i ++ )
78       ptr[ i ] = init.ptr[ i ];    //copy init into object
79 |
80
81 //Destructor for class Array
82 Array::~~Array()
83 |
84   delete [] ptr;                   //reclaim space for array
85   --arrayCount;                    //one fewer objects
86 |
87
88 //get the size of the array
89 int Array::getSize() const { return size; }
90
91 //Overloaded assignment operator
92 //const return avoids: ( a1 = a2 ) = a3
93 const Array &Array::operator=( const Array &right )
94 |
95   if ( &right != this ) { //check for self-assignment
96
97       //for arrays of different sizes, deallocate original
98       //left side array, then allocate new left side array.
99       if ( size != right.size ) {
100           delete [] ptr;           //reclaim space
101           size = right.size;        //resize this object
102           ptr = new int[ size ];    //create space for array copy
103           assert( ptr != 0 );       //terminate if not allocated
104       }
105
106       for ( int i = 0; i < size; i ++ )
107           ptr[ i ] = right.ptr[ i ]; //copy array into object
108   }
109
110   return *this; //enables x = y = z;
111 |
112
113 //Determine if two arrays are equal and
114 //return true, otherwise return false.
115 bool Array::operator==( const Array &right ) const
116 |
117   if ( size != right.size )
118       return false; //arrays of different sizes
119
120   for ( int i = 0; i < size; i ++ )
121       if ( ptr[ i ] != right.ptr[ i ] )
122           return false; //arrays are not equal

```

```
123
124     return true;           //arrays are equal
125 }
126
127 //Overloaded subscript operator for non-const Arrays
128 //reference return creates an lvalue
129 int &Array::operator[] ( int subscript )
130 {
131     //check for subscript out of range error
132     assert( 0 <= subscript && subscript < size );
133
134     return ptr[ subscript ]; //reference return
135 }
136
137 //Overloaded subscript operator for const Arrays
138 //const reference return creates an rvalue
139 const int &Array::operator[] ( int subscript ) const
140 {
141     //check for subscript out of range error
142     assert( 0 <= subscript && subscript < size );
143
144     return ptr[ subscript ]; //const reference return
145 }
146
147 //Return the number of Array objects instantiated
148 //static functions cannot be const
149 int Array::getArrayCount() { return arrayCount; }
150
151 //Overloaded input operator for class Array;
152 //inputs values for entire array.
153 istream &operator >>( istream &input, Array &a )
154 {
155     for ( int i = 0; i < a.size; i ++ )
156         input >> a.ptr[ i ];
157
158     return input; //enables cin >> x >> y;
159 }
160
161 //Overloaded output operator for class Array
162 ostream &operator <<( ostream &output, const Array &a )
163 {
164     int i;
165
166     for ( i = 0; i < a.size; i ++ ) {
167         output << setw( 12 ) << a.ptr[ i ];
168
169         if ( ( i + 1 ) % 4 == 0 ) //4 numbers per row of output
170             output << endl;
171     }
172
173     if ( i % 4 != 0 )
174         output << endl;
```

```

175
176     return output; //enables cout << x << y;
177 }

```

图 8.4 用操作符重载 Array 类——array1.cpp

```

178 //Fig. 8.4: fig08_04.cpp
179 //Driver for simple class Array
180 #include <iostream>
181
182 using std::cout;
183 using std::cin;
184 using std::endl;
185
186 #include "array1.h"
187
188 int main()
189 {
190     //no objects yet
191     cout << "# of arrays instantiated = "
192          << Array::getArrayCount() << '\n';
193
194     //create two arrays and print Array count
195     Array integers1( 7 ), integers2;
196     cout << "# of arrays instantiated = "
197          << Array::getArrayCount() << "\n\n";
198
199     //print integers1 size and contents
200     cout << "Size of array integers1 is "
201          << integers1.getSize()
202          << "\nArray after initialization;\n"
203          << integers1 << '\n';
204
205     //print integers2 size and contents
206     cout << "Size of array integers2 is "
207          << integers2.getSize()
208          << "\nArray after initialization;\n"
209          << integers2 << '\n';
210
211     //input and print integers1 and integers2
212     cout << "Input 17 integers;\n";
213     cin >> integers1 >> integers2;
214     cout << "After input, the arrays contain;\n"
215          << "integers1;\n" << integers1
216          << "integers2;\n" << integers2 << '\n';
217
218     //use overloaded inequality (! =) operator
219     cout << "Evaluating; integers1 != integers2\n";
220     if ( integers1 != integers2 )
221         cout << "They are not equal\n";
222
223     //create array integers3 using integers1 as an

```

```

224 //initializer; print size and contents
225 Array integers3( integers1 );
226
227 cout << "\nSize of array integers3 is "
228     << integers3.getSize()
229     << "\nArray after initialization:\n"
230     << integers3 << '\n';
231
232 //use overloaded assignment ( = ) operator
233 cout << "Assigning integers2 to integers1:\n";
234 integers1 = integers2;
235 cout << "integers1:\n" << integers1
236     << "integers2:\n" << integers2 << '\n';
237
238 //use overloaded equality ( == ) operator
239 cout << "Evaluating; integers1 == integers2\n";
240 if ( integers1 == integers2 )
241     cout << "They are equal\n\n";
242
243 //use overloaded subscript operator to create rvalue
244 cout << "integers1[5] is " << integers1[ 5 ] << '\n';
245
246 //use overloaded subscript operator to create lvalue
247 cout << "Assigning 1000 to integers1[5]\n";
248 integers1[ 5 ] = 1000;
249 cout << "integers1:\n" << integers1 << '\n';
250
251 //attempt to use out of range subscript
252 cout << "Attempt to assign 1000 to integers1[15]" << endl;
253 integers1[ 15 ] = 1000; //ERROR: out of range
254
255 return 0;
256 |

```

#### 输出结果:

```
# of arrays instantiated = 0
```

```
# of arrays instantiated = 2
```

```
Size of array integers1 is 7
```

```
Array after initialization:
```

```

0      0      0      0
0      0      0

```

```
Size of array integers2 is 10
```

```
Array after initialization:
```

```

0      0      0      0
0      0      0      0
0      0

```

```
Input 17 integers:
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

After input, the arrays contain;

integers1:

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 |   |

integers2:

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

Evaluating: integer1 != integers2

They are not equal

Size of array integers3 is 7

Array after initialization;

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 |   |

Assigning integers2 to integers1:

integers1:

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

integers2:

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

Evaluating: integers1 == integers2

They are equal

integers1[5] is 13

Assigning 1000 to integers1[5]

integers1:

|    |       |    |    |
|----|-------|----|----|
| 8  | 9     | 10 | 11 |
| 12 | 1 000 | 14 | 15 |
| 16 | 17    |    |    |

Attempt to assign 1000 to integers1[15]

Assertion failed; 0 <= subscript && subscript < size, file Array1.cpp,  
line 95 abnormal program termination

图 8.4 用操作符重载 Array 类——fig08\_.cpp

Array 类的静态类变量 arrayCount 包含了程序执行过程中实例化的 Array 对象的个数。程序首先使用静态成员函数 getArrayCount(第 192 行)取得了实例化数组的个数。程序实例化了 Array 类的两个对象(第 195 行),对象 integers1 有 7 个元素,对象 integers2 有 10 个元素(默认值由类 Array 的构造函数指定)。第 197 行再次调用 getArrayCount,取得类变量 arrayCount 的值。第 200~203 行用成员函数 getSize 确定 Array integers1 的长度,并使用 Array 重载流插入操作符输出 integers1,以证实构造函数正确地初始化了数组的元素。随后第

206 ~ 209 行的程序输出数组 `integers2` 的长度,并用重载的流插入操作符输出 `integer2` 数组。

接着,程序提示用户输入 17 个整数,Array 重载流读取操作符并使用第 213 行的语句

```
cin >> integers1 >> integers2;
```

把这些值读入两个数组,前 7 个整数保存在 `integers1` 中,其余的保存在 `integers2` 中,为证实输入操作的正确性,程序用流插入操作符输出了这两个数组(第 214 ~ 216 行)。

接下来,程序利用测试条件(第 220 行)

```
integers1 != integers2
```

验证重载的不相等操作符 `!=`,输出结果表明这两个数组确实不相等。

第 225 行程序实例化第 3 个数组 `integers3` 并用数组 `integers1` 对其初始化,这将调用 Array 复制构造函数将 `integers1` 的元素复制到 `integers3`。复制构造函数的详情参见后文。

程序在第 227 ~ 230 行输出 `integers3` 的长度,并使用 Array 重载流插入操作符输出 `integers3`,以证实构造函数正确初始化了数组。

接着,第 234 行通过语句

```
integers1 = integers2;
```

测试重载的赋值操作符 `(=)`。然后打印这两个 Array 验证赋值的正确性(第 235 行和第 236 行)。注意,原来的 `integers1` 中只有 7 个整数,现在必须要使用其能容纳 `integers2` 中 10 个元素的副本。重载的赋值操作符可以改变原来的 `integers1` 的大小,并复制 `integers2` 中的元素。

接下来,第 210 行用重载的相等操作符 `==` 测试赋值后的 `integers1` 和 `integers2` 是否相等。

接下来,第 224 行用重载的下标操作符引用 `integers1[5]`(`integers` 数组范围内的一个元素),这个带下标的数组名作为右值打印 `integers1[5]` 的值,第 248 行将 `integers[5]` 放在赋值语句左边并赋给其一个新值 1 000,注意,operator `[]` 返回引用并作为左值使用(确定 5 是在 `integers1` 的长度范围内)。

第 253 行程序试图将 1 000 赋给 `integers[5]`(越界的元素)。Array 重载了 `[]` 运算捕捉该错误并中止程序。

有趣的是,数组下标操作符不仅可用于数组,还可以用于从其他各种容器类(如链表、字符串等等)中选择元素。此外,下标不仅可以是整数,还可以是字符、字符串、浮点数甚至用户自定义对象。

以上介绍了程序的执行方式。下面再分析一下类的首部和成员函数的定义。第 32 ~ 34 行

```
int size; //size of the array
int *ptr; //pointer to first element of array
static int arrCount; // # of Arrays instantiated
```

是 Array 类的 private 数据成员,包括一个 `int` 类型指针 `ptr`(指向 Array 对象中存储整型的动态分配数组)、一个表示数组元素个数的 `size` 成员以及一个表示已经实例化的数组对象数目的 static 成员 `arrayCount`。

第 12 行和第 13 行

```
friend ostream &operator <<(ostream &, const Array &);
friend istream &operator <<(istream &, Array &);
```

将重载的流插入、流读取操作符声明为类 Array 的友元。当编译器遇到表达式

```
cout << arrayObject
```

时,将生成以下调用来调用

```
operator <<(cout,arrayObject)
```

函数 operator <<(ostream &,const Array &)函数。编译器遇到表达式

```
cin >> arrayObject
```

时,将生成以下调用

```
operator >>(cin,arrayObject)
```

来调用 operator >>(istream &,Array &)函数。因为 Array 对象始终位于流插入操作符和流读取操作符右边,所以这两个操作符函数不能是 Array 的成员函数。如果这些操作符函数是 Array 的成员函数,就必须用难懂的语句

```
arrayObject << cout;
arrayObject >> cin;
```

输入和输出 Array。函数 operator <<(定义见第 162 行)打印由 size 指定的保存在 ptr 中的数组元素的个数,而函数 operator >>(定义见第 153 行)则把数据直接输入到 ptr 所指向的数组中。为了可分别实现连续的输入/输出,这两个操作符分别返回了一个恰当的引用。

第 15 行

```
Array(int = 10); //default constructor
```

声明了类的默认构造函数,并且指定数组元素的默认大小为 10。编译器遇到下列声明

```
Array integers1(7);
```

或与之类似的声明

```
Array integers1 = 7;
```

时,将调用默认构造函数(本例中默认构造函数实际上接收一个 int 参数,默认值为 10)。默认构造函数(定义见第 58 行定义)验证参数并赋值给 size 数据成员,用 new 分配数组所需的内存,将 new 返回的指针赋给数据成员 ptr,然后用 assert 测试 new 操作是否成功,并递增 arrayCount 的值,最后用 for 循环将数组的所有元素初始化为 0。即使没有初始化 Array,也可以在以后读取相应的值,但这样会降低程序的可执行性。Array 和任何对象随时都应保持正确的初始和一致状态。

第 16 行

```
Array(const Array &); //copy constructor
```

声明了一个复制构造函数(定义见第 71 行),它通过建立现有 Array 对象的副本来初始化 Array 对象。必须谨慎对待这种复制操作,避免两个 Array 对象指向同一块动态分配的存储区,默认的成员复制更容易发生这种问题。不论何时需要复制对象时都会调用复制构造函数,如在传值调用时、从被调用函数返回一个对象时、或把某个对象初始化为同类的另外一个对象的副本时。当声明创建类 Array 的一个对象并用另外一个对象对它初始化时,调用复制构造函数。例如,声明

```
Array integers3(integers1);
```

或与之类似的声明

```
Array integers3 = integers1;
```



**常见编程错误 8.6** 注意,复制构造函数时应使用引用调用,而非传值调用,否则复制构造函数调用会造成无穷递归(这是致命的逻辑错误),因为对于传值调用,建立传入复制构造函数的对象副本会造成复制构造函数的递归调用。

复制构造函数 Array 使用成员初始化值将数组的 size 值复制到新数组的数据成员 size 中,用 new 分配新数组所需的内存,把 new 返回的指针赋给数据成员 ptr,然后用 assert 测试 new 操作是否成功,并递增 arrayCount 的值,最后用 for 循环将数组的所有元素作为初始化值复制到新数组中。

**常见编程错误 8.7** 如果构造函数只把源对象的指针复制到目标对象的指针,这两个对象将指向同一块动态分配的内存块,执行析构函数时将释放该内存块,结果导致另一个对象的 ptr 没有定义,这种情况可能会引起运行时错误。

**软件工程知识 8.4** 通常要把构造函数、析构函数、重载的赋值操作符以及复制构造函数一起提供给使用动态内存分配的类。

第 17 行

```
~Array(); // destructor
```

声明了类的析构函数(定义见第 82 行定义)。撤消类 Array 的某个对象时,自动调用析构函数。析构函数用 delete[] 释放构造函数中用 new 动态分配的内存块,然后递减 arrayCount 的值。

第 18 行

```
int getSize() const; // return size
```

声明了读取数组大小的函数。

第 19 行

```
const Array &operator=(const Array &); // assign arrays
```

声明了重载的赋值操作符函数。编译器遇到表达式

```
integers1 = integers2;
```

时,会生成调用

```
integers1.operator=(integers2)
```

来调用 operator = 函数。

成员函数 operator = (定义见第 93 行)测试了这种赋值是否是自我赋值。如果是,就跳过赋值操作(即对象已经是其自身,无需再赋值)。如果不是,成员函数会确定两数组长度是否相同,如果是,则不重新分配左边 Array 对象的原始整数数组。否则成员函数 operator = 将用 delete 释放目标数组原先动态分配的空间,将源数组的数据成员 size 复制到目标数组的 size,用 new 分配目标数组所需的内存并将 new 返回的指针赋给数组的 ptr 成员,用 assert 测试 new 操作是否成功,最后再用 for 循环将源数组的每一个元素复制到目标数组中。不管这种操作是否是自我赋值,成员函数都返回当前对象(即 \*this),这种处理方式允许出现类似 x = y = z 的连续赋值。

**常见编程错误 8.8** 类的对象包含的指向动态分配的内存的指针,但如果不提供重载的赋值操作符和复制的构造函数会造成逻辑错误。

**软件工程知识 8.5** 防止一个类对象被赋值给另一个类对象是可以实现的,具体做法是将赋值操作声明为该对象的 `private` 成员。

**软件工程知识 8.6** 防止类对象被复制是能够实现的;只须令重载的赋值操作符和复制构造函数为 `private` 即可。

第 20 行

```
bool operator==(const Array&) const; //compare equal
```

声明了重载的相等操作符。编译器遇到 `main` 函数中如下表达式

```
integers1==integers2
```

时,编译器将生成如下调用

```
integers1.operator==(integers2)
```

来调用 `operator==` 成员函数。如果数组的 `size` 成员不相等, `operator==` 成员函数就会立即返回 `false`, 否则, 成员函数开始成对比较相应的元素。如果它们全都相等, 则返回 `true`, 一旦发现某一对元素不同时则立即返回 `false`。

第 24 行和第 25 行

```
bool operator!=(const Array&right) const
{return !(*this==right);}
```

定义了重载的不相等操作符(`!=`)。成员函数 `operator!=` 根据重载的相等操作符定义。该函数定义用重载 `operator==` 函数确定一个 `Array` 是否等于另一个 `Array`, 然后返回结果的相反值。这样编写 `operator!=` 函数使程序员可以复用 `operator==` 函数, 减少类中需要编写的代码量。另外, `!=` 的完整函数定义在 `Array` 头文件中, 使编译器可以内联操作符 `!=` 的定义, 消除额外函数调用引起的开销。

第 27 行和第 28 行

```
int&operator[](int); //subscript operator
const int&operator[](int) const; //subscript operator
```

声明了两个重载的下标操作符定义(分别在第 129 行和第 139 行)。编译器遇到 `main` 函数中的表达式

```
integers1[5]
```

时, 将生成调用

```
integers1.operator[](5)
```

来调用重载的 `operator[]` 成员函数。 `constArray` 对象使用下标操作符时, 编译器生成调用 `operator[]` 的常量版本。例如, 如果常量对象 `z` 用语句

```
const Array z(5);
```

初始化, 那么在执行语句

```
cout << z[3] << endl;
```

时, 需要 `operator[]` 的常量版本。常量对象可以只有调用的常量成员函数。

每个 `operator[]` 定义要测试下标是否越界。如果越界, 则程序异常中止。如果没有越界, 则在 `operator[]` 的非常量版本的情况下, 数组的适当元素作为引用返回, 以便使它能用作左值(如用在赋值语句的左边)。而在 `operator[]` 的常量版本的情况下返回右值。

## 第 29 行

```
static int getArrayCount( );    //return count of Arrays
```

声明了静态成员函数 `getArrayCount`。即使不存在 `Array` 类的对象,该成员函数也会返回静态数据成员 `arrayCount` 的值。

## 8.9 类型转换

大多数程序都可处理各种数据类型的信息,有时所有操作会集中于同一种类型。例如,整数加整数仍然是整数(只要结果不是太大,能用整数表示出来)。但是,很多情况下都需要将一种类型的数据转换为另一种类型的数据,赋值、计算、向函数传值以及从函数返回值都可能发生这种情况。对于内部类型,编译器知道如何转换类型。程序员也可以用强制类型转换操作符实现内部类型之间的强制转换。

但是怎样转换用户自定义类型呢?编译器不知道怎样实现用户自定义类型和内部类型之间的转换,程序员必须明确指明。这种转换可以用转换构造函数实现,也就是使用带单个参数的构造函数,这种函数仅仅把其他类型(包括内部类型)的对象转换为某个特定类的对象。本章稍后要用一个转换构造函数将 `char *` 类型的字符串转换为 `String` 类对象。

转换操作符(也称强制类型转换操作符)可以把一种类型的对象转换为其他类的对象或内部类型的对象。这种操作符必须是非静态成员函数,而且不能是友元函数。

### 函数原型

```
A::operator char * ( ) const;
```

声明了一个重载的强制类型转换操作符函数,它根据用户自定义类型 `A` 的对象建立一个临时的 `char *` 类型的对象。重载的强制类型转换操作符函数不能指定返回类型(返回类型是要转换后的对象类型)。如果 `s` 是某个类对象,那么编译器遇到表达式 `(char *)` 时,会产生函数调用 `s.operator char * ( )`,操作数 `s` 是调用成员函数 `operator char *` 的类对象 `s`。

为把用户自定义类型的对象转换为内部类型的对象或类型用户自定义其他对象,可以定义重载的强制类型转换操作符函数。函数原型

```
A::operator int( ) const;
```

```
A::operator otherClass( ) const;
```

声明了两个重载的强制类型转换操作符函数,分别用来把用户自定义类型 `A` 的对象转换为整数和用户自定义类型 `otherClass` 的对象。

强制类型转换操作符和转换构造函数的优点之一是:需要的时候,编译器会自动地调用这些函数建立临时对象。例如,如果用户自定义 `String` 类的某个对象 `s` 出现在程序中需要用 `char *` 类型的位置,如下所示

```
cout << s
```

编译器会调用重载强制类型转换操作符函数 `operator char *` 将对象转换为 `char *`,并在表达式中使用 `char *` 的结果。对 `String` 类提供该转换操作符后,无须重载流插入操作符用 `cout` 输出 `String`。

## 8.10 案例分析:String 类

作为重载的最佳练习,本节要建立一个能处理字符串建立和操作的类(如图 8.5 所示)。String 类已是 C++ 标准库中的一部分,第 19 章将详细介绍 String 类。现在我们用操作符重载建立一个 String 类。

首先我们列出了 String 类的首部,并讨论表示 String 的对象的 private 数据。然后,分析类的 public 接口,讨论该类提供的每一种服务。随后分析 main 函数中的驱动程序。最后讨论了大家都感兴趣的编码风格,也就是用新的 String 类的对象和该类的重载操作符集编写的各种操作符表达式。

```

1  //Fig. 8.5: string1.h
2  //Definition of a String class
3  #ifndef STRING1_H
4  #define STRING1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class String {
12     friend ostream &operator <<( ostream &, const String & );
13     friend istream &operator >>( istream &, String & );
14
15 public:
16     String( const char * = "" ); //conversion/default ctor
17     String( const String & );    //copy constructor
18     ~String();                  //destructor
19     const String &operator =( const String & );    //assignment
20     const String &operator +=( const String & );  //concatenation
21     bool operator! () const;                      //is String empty?
22     bool operator ==( const String & ) const;    //test s1 == s2
23     bool operator <( const String & ) const;     //test s1 < s2
24
25     //test s1 != s2
26     bool operator !=( const String & right ) const
27     { return ! ( *this == right ); }
28
29     //test s1 > s2
30     bool operator >( const String &right ) const
31     { return right < *this; }
32
33     //test s1 <= s2
34     bool operator <=( const String &right ) const
35     { return ! ( right < *this ); }
36
37     //test s1 >= s2

```

```

38  bool operator >=( const String &right ) const
39      { return ! ( *this < right ); }
40
41  char &operator[]( int );           //subscript operator
42  const char &operator[]( int ) const; //subscript operator
43  String operator()( int, int );     //return a substring
44  int getLength() const;            //return string length
45
46  private:
47      int length;                   //string length
48      char *sPtr;                   //pointer to start of string
49
50  void setString( const char * ); //utility function
51  };
52
53  #endif

```

图 8.5 用操作符重载 String 类——string1.h

```

54  //Fig. 8.5: string1.cpp
55  //Member function definitions for class String
56  #include <iostream>
57
58  using std::cout;
59  using std::endl;
60
61  #include <iomanip>
62
63  using std::setw;
64
65  #include <cstring>
66  #include <cassert>
67  #include "string1.h"
68
69  //Conversion constructor: Convert char * to String
70  String::String( const char *s ) : length( strlen( s ) )
71  {
72      cout << "Conversion constructor: " << s << '\n';
73      setString( s );           //call utility function
74  }
75
76  //Copy constructor
77  String::String( const String &copy ) : length( copy.length )
78  {
79      cout << "Copy constructor: " << copy.sPtr << '\n';
80      setString( copy.sPtr ); //call utility function
81  }
82
83  //Destructor
84  String::~String()
85  {
86      cout << "Destructor: " << sPtr << '\n';

```

```

87  delete [] sPtr; //reclaim string
88  |
89
90  //Overloaded = operator; avoids self assignment
91  const String &String::operator =( const String &right )
92  {
93      cout << "operator = called\n";
94
95      if ( &right != this ) {          //avoid self assignment
96          delete [] sPtr;              //prevents memory leak
97          length = right.length;      //new String length
98          setString( right.sPtr );    //call utility function
99      |
100     else
101         cout << "Attempted assignment of a String to itself\n";
102
103     return *this; //enables cascaded assignments
104 |
105
106 //Concatenate right operand to this object and
107 //store in this object.
108 const String &String::operator +=( const String &right )
109 {
110     char *tempPtr = sPtr;            //hold to be able to delete
111     length += right.length;          //new String length
112     sPtr = new char[ length + 1 ];    //create space
113     assert( sPtr != 0 );              //terminate if memory not allocated
114     strcpy( sPtr, tempPtr );          //left part of new String
115     strcat( sPtr, right.sPtr );      //right part of new String
116     delete [] tempPtr;               //reclaim old space
117     return *this;                   //enables cascaded calls
118 |
119
120 //Is this String empty?
121 bool String::operator! () const { return length == 0; |
122
123 //Is this String equal to right String?
124 bool String::operator ==( const String &right ) const
125     { return strcmp( sPtr, right.sPtr ) == 0; |
126
127 //Is this String less than right String?
128 bool String::operator <( const String &right ) const
129     { return strcmp( sPtr, right.sPtr ) < 0; |
130
131 //Return a reference to a character in a String as an lvalue.
132 char &String::operator[]( int subscript )
133 |
134     //First test for subscript out of range
135     assert( subscript >= 0 && subscript < length );
136
137     return sPtr[ subscript ]; //creates lvalue

```

```
138 |
139
140 //Return a reference to a character in a String as an rvalue.
141 const char &String::operator[]( int subscript ) const
142 {
143     //First test for subscript out of range
144     assert( subscript >= 0 && subscript < length );
145
146     return sPtr[ subscript ]; //creates rvalue
147 }
148
149 //Return a substring beginning at index and
150 //of length subLength
151 String String::operator()( int index, int subLength )
152 {
153     //ensure index is in range and substring length >= 0
154     assert( index >= 0 && index < length && subLength >= 0 );
155
156     //determine length of substring
157     int len;
158
159     if ( ( subLength == 0 ) || ( index + subLength > length ) )
160         len = length - index;
161     else
162         len = subLength;
163
164     //allocate temporary array for substring and
165     //terminating null character
166     char *tempPtr = new char[ len + 1 ];
167     assert( tempPtr != 0 ); //ensure space allocated
168
169     //copy substring into char array and terminate string
170     strncpy( tempPtr, &sPtr[ index ], len );
171     tempPtr[ len ] = '\0';
172
173     //Create temporary String object containing the substring
174     String tempString( tempPtr );
175     delete [] tempPtr; //delete the temporary array
176
177     return tempString; //return copy of the temporary String
178 }
179
180 //Return string length
181 int String::getLength() const { return length; }
182
183 //Utility function to be called by constructors and
184 //assignment operator.
185 void String::setString( const char *string2 )
186 {
187     sPtr = new char[ length + 1 ]; //allocate storage
188     assert( sPtr != 0 ); //terminate if memory not allocated
```

```

189 strcpy( sPtr, string2 ); //copy literal to object
190 {
191
192 //Overloaded output operator
193 ostream &operator <<( ostream &output, const String &s )
194 {
195
196 output << s.sPtr;
197
198 return output; //enables cascading
199 |
200 //Overloaded input operator
201 istream &operator >>( istream &input, String &s )
202 {
203
204 char temp[ 100 ]; //buffer to store input
205
206 input >> setw( 100 ) >> temp;
207 s = temp; //use String class assignment operator
208 return input; //enables cascading
209 }

```

图 8.5 用操作符重载 String 类——string1.cpp

接下来讨论 String 类的成员函数定义。对于每个重载的操作符函数,驱动程序都有调用重载操作符的代码,以及针对这些重载操作符函数工作原理的解释。

```

208 //Fig. 8.5; fig08_05.cpp
209 //Driver for class String
210 #include <iostream>
211
212 using std::cout;
213 using std::endl;
214
215 #include "string1.h"
216
217 int main()
218 {
219     String s1( "happy" ), s2( " birthday" ), s3;
220
221     //test overloaded equality and relational operators
222     cout << "s1 is \" " << s1 << "\"; s2 is \" " << s2
223         << "\"; s3 is \" " << s3 << "\"
224         << " \nThe results of comparing s2 and s1:"
225         << " \ns2 == s1 yields "
226         << ( s2 == s1 ? "true" : "false" )
227         << " \ns2 != s1 yields "
228         << ( s2 != s1 ? "true" : "false" )
229         << " \ns2 > s1 yields "
230         << ( s2 > s1 ? "true" : "false" )

```



```

231     << "s2 < s1 yields "
232     << ( s2 < s1 ? "true" : "false" )
233     << "s2 >= s1 yields "
234     << ( s2 >= s1 ? "true" : "false" )
235     << "s2 <= s1 yields "
236     << ( s2 <= s1 ? "true" : "false" );
237
238 //test overloaded String empty (!) operator
239 cout << "Testing ! s3:\n";
240 if ( ! s3 ) {
241     cout << "s3 is empty; assigning s1 to s3;\n";
242     s3 = s1; //test overloaded assignment
243     cout << "s3 is \"" << s3 << "\"";
244 }
245
246 //test overloaded String concatenation operator
247 cout << "s1 += s2 yields s1 = ";
248 s1 += s2; //test overloaded concatenation
249 cout << s1;
250
251 //test conversion constructor
252 cout << "s1 += ' to you' yields\n";
253 s1 += " to you"; //test conversion constructor
254 cout << "s1 = " << s1 << "\n";
255
256 //test overloaded function call operator () for substring
257 cout << "The substring of s1 starting at\n"
258     << "location 0 for 14 characters, s1(0, 14), is:\n"
259     << s1( 0, 14 ) << "\n";
260
261 //test substring "to-end-of-String" option
262 cout << "The substring of s1 starting at\n"
263     << "location 15, s1(15, 0), is: "
264     << s1( 15, 0 ) << "\n"; //0 is "to end of string"
265
266 //test copy constructor
267 String *s4Ptr = new String( s1 );
268 cout << "s4Ptr = " << *s4Ptr << "\n";
269
270 //test assignment ( = ) operator with self-assignment
271 cout << "assigning *s4Ptr to *s4Ptr\n";
272 *s4Ptr = *s4Ptr; //test overloaded assignment
273 cout << "s4Ptr = " << *s4Ptr << "\n";
274
275 //test destructor
276 delete s4Ptr;
277
278 //test using subscript operator to create lvalue
279 s1[ 0 ] = 'H';
280 s1[ 6 ] = 'B';
281 cout << "s1 after s1[0] = 'H' and s1[6] = 'B' is: "

```

```

282         << s1 << "\n\n";
283
284     //test subscript out of range
285     cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
286     s1[ 30 ] = 'd'; //ERROR: subscript out of range
287
288     return 0;
289 }

```

输出结果:

```

Conversion constructor: happy
Conversion constructor: birthday
Conversion constructor:
s1 is "happy"; s2 is "birthday"; s3 is " "
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3;
s3 is empty; assigning s1 to s3;
operator = called
s3 is "happy"
s1 += s2 yields s1 = happy birthday

s1 += "to you" yields
Conversion constructor: to you
Conversion constructor: to you
s1 = happy birthday to you

Conversion constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters,s1(0,14),is:
happy birthday

Destructor: happy birthday
Conversion constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15,s1(15,0),is: to you

Destructor: to you
Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

```

```

assigning *s4Ptr to * s4Ptr
operator = called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:

Assertion failed: subscript >= 0 && subscript < length, file
string1.cpp, line 82

Abnormal program termination

```

图 8.5 用操作符重载 String 类——fig08\_05.cpp

我们首先介绍 String 的内部表达式。第 47 行和第 48 行

```

int length; //string length
char *sPtr; //pointer to start of string

```

声明了类的 private 数据成员。String 的对象有一个 length 域,表明字符串中除字符串中止符以外的字符个数和指向动态分配内存(表示字符串)的指针 sPtr。

接下来看看图 8.5 中定义 String 类的头文件。第 12 行和第 13 行

```

friend ostream &operator <<(ostream &,const String &);
friend istream &operator >>(istream &,string &);

```

声明重载的流插入操作符函数 operator << (定义见第 193 行)和流读取操作符函数 operator >> (定义见第 200 行)为类的友元。这两个函数的实现简洁明了。

第 16 行

```

String(const char * = ""); //conversion/default ctor

```

声明了一个转换构造函数。该构造函数(定义见第 70 行)有一个 const char \* 类型的参数(默认值是空字符串)。该函数实例化了 String 的一个对象,该对象包含了与参数相同的字符串。任何只带一个参数的构造函数都可以认为是一种转换构造函数。稍后会提到,转换构造函数非常适用于用 char \* 参数对 String 类执行任何操作。转换构造函数把一个 char \* 字符串转换为 String 的对象(然后该对象要赋给目标 String 对象)。使用这种转换构造函数意味着不必再为将字符串赋给 String 类对象提供重载的赋值操作符,编译器先调用该函数建立包含该字符串的一个临时 String 对象,然后再调用重载的赋值操作符将临时 String 对象赋给另一个 String 对象。

**软件工程知识 8.7** 用转换构造函数实现隐式转换时,C++ 只会用一个隐式构造函数调用来尝试满足重载赋值操作符的需要。通过执行一系列隐式的、用户自定义的类型转换来满足重载操作符的需要是不可能的。

在指定类似 String s1(“happy”)的声明时,调用 String 的转换构造函数。转换构造函数计算了字符串的长度并将该长度赋给 private 数据成员 length,然后调用 private 工具函数 setString。函数 setString(定义见第 185 行)用 new 为 private 数据成员 sPtr 分配足够多的空间,

并用 `assert` 测试内存分配操作是否成功。如果成功,则用函数 `strcpy` 把字符串复制到对象中。

第 17 行

```
String( const String &);           //copy constructor
```

是一个构造函数(定义见第 77 行),它通过复制现有的 `String` 对象来初始化一个 `String` 对象。这种复制操作必须谨慎,避免使两个 `String` 对象指向同一块动态分配的内存区,默认的成员复制更容易出现这种问题。复制构造函数除了将源 `String` 对象的 `length` 成员复制到目标 `String` 对象外,其余操作和转换构造函数类似。注意,复制构造函数为目标对象的内部字符串分配了新的存储空间,如果它只是简单地将源对象中的 `sPtr` 复制到目标对象的 `sPtr`,这两个对象就会指向同一块动态分配的内存块。执行一个对象的析构函数将释放该内存块,从而使另一个对象的 `sPtr` 没有定义,这可能引起严重的运行时错误。

第 18 行

```
~String();           // destructor
```

声明了 `String` 类的析构函数(定义见第 84 行)。该析构函数用 `delete` 回收构造函数中由 `new` 为字符串分配的动态内存。

第 19 行

```
const String &operator =(const String &); // assignment
```

声明了重载的赋值操作符函数 `operator =` (定义见第 91 行)。编译器在遇到类似 `string1 = string2` 的表达式时,就会生成函数调用:

```
string1.operator =(string2);
```

重载的赋值操作符函数 `operator =` 测试了这种赋值是否为自我赋值。如果是自我赋值运算,函数就会由于该对象已存在而简单返回。如果忽略自我赋值测试,函数就会立即释放目标对象所占用的空间,将源对象中的 `length` 字段复制到目标对象并调用 `setString` (第 185 行),为目标对象建立新空间,确定 `new` 操作是否成功,最后用函数 `strcpy` 将源对象的字符串复制到目标对象中。不管上述赋值是否为自我赋值,都返回 `*this` 以确保连续赋值。

第 20 行

```
const String &operator +=(const String &); // concatenation
```

声明了重载的字符串连接操作符(定义见第 108 行)。编译器遇到 `main` 函数中的表达式 `s1 += s2` 时,会生成函数调用 `s1.operator +=(s2)`。函数 `operator +=` 生成一个临时指针,用以存放当前对象的字符串指针,直到可以释放该字符串的内存为止,该函数还计算了连接后的字符串长度,用 `new` 为字符串分配空间,用 `assert` 测试 `new` 操作是否成功。如果成功,就用函数 `strcpy` 将原来的字符串复制到分配的空间中,然后用函数 `strcat` 将源对象的字符串连接到所分配的空间中,最后再用 `delete` 释放该对象原来的字符串占据的空间,返回 `*this` 作为 `String &` 以确保操作符 `+=` 可连续执行。

连接 `String` 类型的对象和 `char *` 类型的对象无须另行重载一个连接操作符, `const char *` 转换构造函数将传统的字符串转换为临时的 `String` 类型的对象,然后由该对象匹配现有的重载连接操作符。C++ 为实现匹配只能在同一层内执行这样的转换。在执行内部类型和类之间的转换前,C++ 还能在内部类型之间执行编译器隐式定义的类型转换。注意,生成临时

String 对象时,调用转换构造函数和析构函数(参见图 8.5 中 `s1 += "to you"` 产生的输出结果)。这是隐式转换期间生成和删除临时类对象时向类客户隐藏的函数调用开销的一个例子。复制构造函数按值调用传递参数和按值返回类对象时也会产生类似开销。

**性能提示 8.2** 与先执行隐式类型转换然后再执行连接操作相比,令重载的连接操作符 `+=` 只有一个 `const char *` 类型参数,执行效率更高。隐式类型转换需要较少的代码,也较少出错。

第 21 行

```
bool operator! ( ) const;    // is String empty?
```

声明了重载的非操作符(定义第 121 行)。该操作符通常与字符串类一起使用,测试字符串是否为空。例如,当编译器遇到表达式 `! string1` 时,就会生成函数调用

```
string1.operator! ( )
```

该函数只返回 `length` 是否等于 0 的测试结果。

语句

```
bool operator ==(const String &) const; //test s1 == s2
```

```
bool operator <(const String &) const; //test s1 < s2
```

为 `string` 类声明了重载的相等操作符(定义见第 124 行)和关系操作符(定义见第 128 行)。其工作原理也类似,因此我们只以重载的操作符 `=` 为例。编译器在遇到表达式 `string1 == string2` 时,会生成下列函数调用

```
string1.operator ==(string2)
```

如果 `string1` 等于 `string2` 时,则返回 `true`。上述操作符都用函数 `strcmp` 比较 `String` 对象中的字符串。注意我们使用 `<cstring>` 中的函数 `strcmp`。许多 C++ 程序员提倡用一些重载操作符函数实现另外一些重载操作符函数,因此 `!=`、`>`、`<=` 和 `>=` 操作符都可以用 `operator ==` 和 `operator <` 来实现(第 26 ~ 39 行)。例如,重载函数 `operator >=` 在头文件中的实现(第 38 行)

```
bool operator >=(const String &right) const
```

```
{return ! ( *this < right );}
```

上述 `operator >=` 定义用重载的操作符 `<` 确定 `String` 对象是否大于或等于另一个 `String` 对象。注意 `!=`、`>`、`<=` 和 `>=` 操作符函数都在头文件中定义。编译器内联这些定义,消除了额外函数调用的开销。

**软件工程知识 8.8** 利用以前定义的成员函数实现成员函数,程序员可重用代码,减少了编写代码工作量。

第 41 行和第 42 行

```
char &operator[ ] ( int ); // subscript operator
```

```
const char &operator[ ] ( int ) const; // subscript operator
```

声明了两个重载的下标操作符(定义见第 132 行和第 141 行)。一个用于常量 `String`,一个用于非常量。编译器遇到 `string[10]` 这样的表达式时,会生成函数调用 `string1.operator[ ](0)` (根据 `String` 是否为常量类型而使用相应的 `operator[ ]` 版本)。函数 `operator[ ]` 首先用 `assert` 检查下标范围。如果下标越界,就打印一条出错信息并使程序异常中止。如果下标没有越界,非常量版本的 `operator[ ]` 就会返回一个 `char &` 类型的值,它是对 `String` 对象相应字符的

引用,可用作左值,修改 String 对象中指定的字符。常量版本的 `operator[]` 则返回 String 对象的相应字符,这里 `char &` 可以作为右值,读取该字符值。

**测试和调试提示 8.1** 从 String 类的重载下标操作符返回 `char` 引用是危险的。例如,客户代码可以用这个引用在字符串类的任何位置插入空中字符(`'\0'`)。

第 43 行

```
String operator()(int,int); // return a substring
```

声明了重载的函数调用操作符(定义见第 151 行)。在 String 类中,为了从 String 对象中选择一个子串,经常需要重载该操作符。两个整数参数指定了所选定子串的起始位置和长度。如果起始位置越界或者子串长度为负,则发出错误信息。如果子串长度为 0,则选择的子串为从选定的开始位置一直到 String 对象的末尾。例如,如果 `string1` 是一个包含字符串“AEIOU”的 String 对象,编译器遇到表达式 `string1(2,2)` 时,就会生成函数调用 `string1.operator(2,2)`。执行该函数调用时,产生并返回一个包含字符串“IO”的动态分配的新 String 对象。

因为函数可能有一个冗长而复杂的参数表,所以重载的函数调用操作符()`()` 可具有强大的功能,可以完成很多有意义的操作。函数调用操作符的另一个用法是用作数组的下标符号。例如,有的程序员不喜欢用 C 的两个方括号表示二维数组(如 `a[b][c]`),他们更偏爱重载函数调用操作符,用 `a(b,c)` 表示二维数组。重载函数调用操作符只能是非静态成员函数。只有当“函数名”是 String 类的对象时,才能使用该操作符。

第 44 行

```
inline int getLength() const; // return string length
```

声明了返回 String 对象长度的函数。注意,该函数(定义见第 181 行)通过返回 String 类的 `private` 数据值获得字符串的长度。

鉴于此,大家现在应查看 `main` 函数的代码,查看输出结果,了解重载操作符的各种用法。

## 8.11 重载 ++ 和 --

所有自增和自减操作符(即前置和后置的自增及自减操作符)都可以被重载。本节介绍编译器如何识别前置和后置的自增及自减操作符。

要重载既允许前置又允许后置的自增操作符,每个重载的操作符函数都必须有一个明确的特征以使编译器能确定采用哪个 ++ 版本。重载前置的方法与重载其他前置一元操作符一样。

例如:如果要为 Date 对象 `d1` 增加一天。编译器在遇到前置自增表达式

```
++d1
```

时,会生成成员函数调用

```
d1.operator ++()
```

该函数的原型为

```
Date &operator ++();
```

如果前置自增操作符函数是一个非成员函数,那么编译器在遇到表达式

```
++d1
```

时,就会生成函数调用

```
operator ++(d1)
```

该函数的函数原型在类 Date 中的声明形式为

```
friend Date &operator ++(Date &);
```

由于编译器必须区分重载的前置和后置自增操作符函数,所以重载后置自增操作符出现了问题。C++ 采用的解决办法是,编译器在遇到后置自增表达式

```
d ++
```

时,生成成员函数调用

```
d1.operator ++(0)
```

该函数的函数原型为

```
Date operator ++(int)
```

严格地说,0 是一个伪值,它使操作符函数 operator ++ 在用于后置自增操作和前置自增操作时的参数表有所区别。

如果后置自增操作符函数是一个非成员函数,那么编译器在遇到表达式

```
d1 ++
```

时,会生成函数调用

```
operator ++(d1,0)
```

该函数的原型为

```
friend Date operator ++(Date &,int);
```

再次提醒大家注意,编辑器使用的是参数 0,以便于区分后置自增操作所用的 operator ++ 参数列表和前置自增操作的参数表。

本节讲述的重载前置和后置自增操作符的内容同样可用于重载前置和后置自减操作符。下一节将分析使用了重载前置和后置自增操作符的 Date 类。

## 8.12 案例分析:Date 类

图 8.6 演示了 Date 类。Date 类用重载的前置和后置自增操作符将一个 Date 对象增加 1 天,必要时递增年、月。

```
1 //Fig. 8.6: date1.h
2 //Definition of class Date
3 #ifndef DATE1_H
4 #define DATE1_H
5 #include <iostream>
6
7 using std::ostream;
8
9 class Date {
10     friend ostream &operator <<( ostream &, const Date & );
11 }
```

```

12 public:
13     Date( int m = 1, int d = 1, int y = 1900 ); //constructor
14     void setDate( int, int, int ); //set the date
15     Date &operator ++(); //preincrement operator
16     Date operator ++( int ); //postincrement operator
17     const Date &operator +=( int ); //add days, modify object
18     bool leapYear( int ) const; //is this a leap year?
19     bool endOfMonth( int ) const; //is this end of month?
20
21 private:
22     int month;
23     int day;
24     int year;
25
26     static const int days[]; //array of days per month
27     void helpIncrement(); //utility function
28 |;
29
30 #endif

```

图 8.6 重载自增操作符 Date 类——date1.h

```

31 //Fig. 8.6: date1.cpp
32 //Member function definitions for Date class
33 #include <iostream>
34 #include "date1.h"
35
36 //Initialize static member at file scope;
37 //one class-wide copy.
38 const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
39                             31, 31, 30, 31, 30, 31 };
40
41 //Date constructor
42 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
43
44 //set the date
45 void Date::setDate( int mm, int dd, int yy )
46 {
47     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
48     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
49
50     //test for a leap year
51     if ( month == 2 && leapYear( year ) )
52         day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
53     else
54         day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
55 }
56
57 //Preincrement operator overloaded as a member function.
58 Date &Date::operator ++()
59 {
60     helpIncrement();

```



```
61     return *this; //reference return to create an lvalue
62 }
63
64 //Postincrement operator overloaded as a member function.
65 //Note that the dummy integer parameter does not have a
66 //parameter name.
67 Date Date::operator ++( int )
68 {
69     Date temp = *this;
70     helpIncrement();
71
72     //return non-incremented, saved, temporary object
73     return temp; //value return; not a reference return
74 }
75
76 //Add a specific number of days to a date
77 const Date &Date::operator +=( int additionalDays )
78 {
79     for ( int i = 0; i < additionalDays; i ++ )
80         helpIncrement();
81
82     return *this;    //enables cascading
83 }
84
85 //If the year is a leap year, return true;
86 //otherwise, return false
87 bool Date::leapYear( int y ) const
88 {
89     if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
90         return true; //a leap year
91     else
92         return false; //not a leap year
93 }
94
95 //Determine if the day is the end of the month
96 bool Date::endOfMonth( int d ) const
97 {
98     if ( month == 2 && leapYear( year ) )
99         return d == 29; //last day of Feb. in leap year
100     else
101         return d == days[ month ];
102 }
103
104 //Function to help increment the date
105 void Date::helpIncrement()
106 {
107     if ( endOfMonth( day ) && month == 12 ) { //end year
108         day = 1;
109         month = 1;
110         ++year;
111     }
112     else if ( endOfMonth( day ) ) { //end month
```

```

113     day = 1;
114     ++month;
115 }
116 else    //not end of month or year; increment day
117     ++day;
118 }
119
120 //Overloaded output operator
121 ostream&operator <<( ostream&output, const Date &d )
122 {
123     static char *monthName[ 13 ] = { "", "January",
124     "February", "March", "April", "May", "June",
125     "July", "August", "September", "October",
126     "November", "December" };
127
128     output << monthName[ d.month ] << "
129         << d.day << ", " << d.year;
130
131     return output; //enables cascading
132 }

```

图 8.6 重载自增操作符 Date 类——date1.cpp

```

133 //Fig. 8.6: fig08_06.cpp
134 //Driver for class Date
135 #include <iostream>
136
137 using std::cout;
138 using std::endl;
139
140 #include "date1.h"
141
142 int main()
143 {
144     Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
145     cout << "d1 is " << d1
146         << "\nd2 is " << d2
147         << "\nd3 is " << d3 << "\n\n";
148
149     cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
150
151     d3.setDate( 2, 28, 1992 );
152     cout << " d3 is " << d3;
153     cout << "\n ++d3 is " << ++d3 << "\n\n";
154
155     Date d4( 3, 18, 1969 );
156
157     cout << "Testing the preincrement operator;\n"
158         << " d4 is " << d4 << '\n';
159     cout << " ++d4 is " << ++d4 << '\n';
160     cout << " d4 is " << d4 << "\n\n";
161

```

```

162  cout << "Testing the postincrement operator:\n"
163      << "d4 is " << d4 << '\n';
164  cout << "d4 ++ is " << d4 ++ << '\n';
165  cout << "d4 is " << d4 << endl;
166
167  return 0;
168  |

```

输出结果:

```

d1 is January 1, 1990
d2 is December 27, 1992
d3 is January 1, 1990

```

```

d2 += 7 is January 3, 1993

```

```

    d3 is February 28, 1992
++ d3 is February 29, 1992

```

Testing the preincrement operator;

```

    d4 is March 18, 1969
++d4 March 19, 1969
    d4 is March 19, 1969

```

Testing the postincrement operator:

```

    d4 is March 19, 1969
d4 ++ is March 19, 1969
    d4 is March 20, 1969

```

图 8.6 重载自增操作符 Date 类——fig08\_06.cpp

Date 类的 public 接口包含以下成员函数:一个重载的流插入操作符、一个默认的构造函数、一个 setDate 函数、一个重载的前置自增操作符函数、一个重载的后置自增操作符函数、一个重载的加法赋值操作符(+=)、一个检测闰年的函数和一个判断是否为每月最后一天的函数。

Main 函数中的驱动程序生成了几个日期对象,它们是:初始化为 1990 年 1 月 1 日的 d1,初始化为 1992 年 12 月 27 日的 d2 以及初始化一个非法日期的 d3。Date 的构造函数调用函数 setDate 检测月、日和年的合法性。如果月非法则置为 1,年非法则置为 1900,日期非法则置为 1。

驱动程序用重载的流插入操作符输出建立的每个 Date 对象。程序用重载的操作符 += 将对象 d2 增加 7 天,用函数 setDate 将对象 d3 设置为 1992 年 2 月 28 日,接着将一个新对象 d4 设置为 1969 年 3 月 18 日,并用重载的前置自增操作符将 d4 增加 1 天。为证实执行过程的正确性,在执行前置自增操作的前后分别输出了日期。最后,用重载的后置自增操作符将对象 d4 增加一天。为了证实执行过程的正确性,在执行后置自增操作的前后也分别输出了日期。

重载前置自增操作符简单明了,前置自增操作符调用 private 工具函数 helpIncrement 来执行实际的自增运算。该函数要处理日期的“边界”和“进位”情况,这往往出现在需要递增

每月的最后一天时。此时需要将递增月份,如果月份已经是12,则必须递增年份。函数 `helpIncrement` 用函数 `leapYear` 和 `end of Month` 正确递增日期。

重载的前置自增操作符返回对当前对象 `Date` (已自增)的引用。这是因为当前对象的 `* this` 已作为 `Date&` 返回。

重载后置自增操作符需要一点技巧。为模拟后置操作,函数必须返回该 `Date` 对象自增之前的副本。在进入 `operator ++` 时,函数先把当前对象 (`* this`) 保存在 `temp` 中,然后调用 `helpIncrement` 递增当前的 `Date` 对象,最后返回未递增的对象在 `temp` 中的副本。注意这个函数不能返回对局部 `Date` 对象 `temp` 的引用,因为声明该对象的函数在退出时删除了局部变量。所以,声明这个函数的返回类型为 `Date&` 将返回对已不存在的对象的引用。返回局部变量的引用是常见错误,一些编译器会发出警告消息。

## 8.13 小结

- 操作符 `<<` 在 C++ 中用途较广,既可用作流插入操作符又可用作左移位操作符,这是操作符重载的典型示例。同样地,操作符 `>>` 也是 C++ 中的重载操作符,它既可用作流读取操作符,也可用作右移位操作符。
- C++ 允许程序员重载大多数操作符,使其更符合使用场景。编译器根据操作符的使用方式生成合适的代码。
- 操作符重载提高了 C++ 的可扩展性。
- 操作符重载是通过编写函数定义实现的。函数名由关键字 `operator` 和其后要重载的操作符组成。
- 用于类的对象的操作符必须重载,但是有两种情况例外。对于类型相同的两个对象使用赋值操作符而不用重载,默认的方式是复制数据成员。地址操作符 (`&`) 无需重载也可用于任何类的对象,它返回对象在内存中的地址。
- C++ 为其内部类型提供了丰富的操作符集,重载这些操作符的目的是为用户自定义的类型提供同样简洁的表达式。
- 重载不能改变操作符的优先级和结合性。
- 重载不能改变操作符操作数的个数。重载的一元操作符仍然是一元操作符,重载的二元操作符仍然是二元操作符。C++ 惟一的三元操作符 (`?:`) 不能重载。
- 不能建立新的操作符符号,只有现有的操作符才能重载。
- 操作符重载会改变该操作符用于内部类型的对象时的含义。
- 在重载操作符 (`()`, `[]`, `->` 或者 `=` 时,操作符重载函数必须声明为类的一个成员。
- 操作符函数既可以是成员函数,也可以是非成员函数。
- 当操作符函数是一个成员函数时,最左边的操作数必须是操作符类的一个类对象 (或者对该类对象的引用)。
- 如果左边的操作数必须是一个不同的类的对象,该操作符函数必须作为一个非成员函数来实现。
- 只有当二元操作符的最左边的操作数是该类的一个对象或者当一元操作符的操作

数是该类的一个对象时,才会调用操作符成员函数。

- 选择非成员函数重载操作符的另一个原因是使操作符具有可交换性。例如,指定正确的重载操作符定义,操作符左边的参数可以充当其他数据成员的对象。
- 类的一元操作符可重载为无参数的非 static 成员函数或者带有一个参数的非成员函数,参数必须是用户自定义类型的对象或者对该的对象引用。
- 二元操作符可以重载为带有一个参数的非 static 成员函数或者带有两个参数的非成员函数(参数之一必须是类的对象或者是对类的对象的引用)。
- 数组下标操作符不仅可用于数组,还可以用于从其他各种顺序容器类(如链表、字符串等)中选择元素。此外,下标不仅可以是整数,还可以是字符或者字符串等。
- 复制构造函数根据同类中的其他对象初始化一个对象。不论何时需要复制对象时都会调用复制构造函数,例如在按值调用时,或是从被调用函数返回值时。在复制构造函数中,被复制的对象是通过引用传递的。
- 编译器不知道怎样实现用户自定义类型和内部类型之间的转换,程序员必须明确指明。这种转换可以用转换构造函数实现(即带有单个参数的构造函数),这种函数仅把其他类型的对象转换为某个特定类的对象。
- 转换操作符(又称为强制类型转换操作符)可以将一类的对象转换为其他类对象或内部类型的对象。这种操作符必须是一个非静态成员函数,而不能是友元函数。
- 转换构造函数是带有一个参数的构造函数,用于将参数转换为构造函数所在类的对象。编译器可隐式调用这种构造函数。
- 赋值操作符是最常用的重载操作符,通常用来把一个对象赋给同类的另一个对象。通过使用转换构造函数,赋值操作符也能够实现不同类之间对象的相互赋值。
- 在不提供重载的赋值操作符时,赋值操作符的默认行为是复制类的数据成员。在有些情况下这是允许的,但是当对象中包含指向动态分配的内存区的指针时,成员复制会导致两个不同的对象指向同一块动态分配的内存区。这样,调用其中一个对象的析构函数将释放该动态分配的内存块,如果另一个对象引用该内存区,其结果会不确定。
- 要重载既能允许前置以允许后置的自增操作符,每个重载的操作符函数必须有一个明确的特征,以使编译确定要使用的C++ 版本。重载前置 ++ 的方法与重载其他前置一元操作符一样。向后置自增操作符函数提供第二个参数(必须是 int)类型以区分前置和后置自增操作符函数。实际上,用户并没有为该特定的整数参数提供值,它只是让编译器区分前置和后置自增操作符函数。

## 本章术语

Array class    Array 类

cascaded overloaded operators    连续使用重载操作符

cast operator function    强制类型转换操作符函数

conversion between built-in types and classes

类和内部类型之间的转换

conversion between class types    类类型之间的转换

conversion constructor    转换构造函数

conversion function    转换函数

conversion function    转换操作符

copy constructor    复制构造函数

default memberwise copy    默认按位成员复制

dangling pointer    悬挂指针

Date class Date 类

explicit type conversion (with casts)

显式类型转换(使用强制类型转换操作符)

friend overloaded operator function

友元重载操作符函数

function call operator 函数调用操作符

HugeInteger class HugeInteger 类

implicit type conversions 隐式类型转换

member function overloaded operator

成员函数重载操作符

memory leak 内存泄漏

non-overloadable operators 非可重载操作符

operator keyword 操作符关键字

operator overloading 操作符重载

operators implemented as functions

将操作符实现为函数

overloadable operators 可重载操作符

overloaded != operator 重载的 != 操作符

overloaded + operator 重载的 + 操作符

overloaded ++ operator 重载的 ++ 操作符

overloaded-operator 重载的 - 操作符

overloaded < operator 重载的 < 操作符

overloaded << operator 重载的 << 操作符

overloaded <= operator 重载的 <= 操作符

overloaded == operator 重载的 == 操作符

overloaded > operator 重载的 > 操作符

overloaded >= operator 重载的 >= 操作符

overloaded >> operator 重载的 >> 操作符

overloaded assignment(=) operator

重载的赋值(=)操作符

overloaded a binary operator 重载二元操作符

overloaded a unary operator 重载一元操作符

overloaded [] operator 重载的 [] 操作符

overloading 重载

PhoneNumber class PhoneNumber 类

prefix unary operator overloading

前置一元操作符重载

posifix unary operator overloading

后置一元操作符重载

self assignment 自我赋值

single-argument constructor 单个参数构造函数

String class String 类

string concatenation 字符串连接

substring 子串

user-defined conversion 用户自定义转换

user-defined type 用户自定义类型

## 常见编程错误

8.1 试图重载不能重载的操作符是语法错误。

8.2 试图创建新的操作符是语法错误。

8.3 试图改变操作符对内部类型的对象的作用方式是语法错误。

8.4 误以为重载了某个操作符(如“+”)可以自动重载相关的操作符(如“+=”),重载了“==”就自动重载了“!=”。操作符只能显式重载(不会隐式重载)。

8.5 试图通过操作符重载改变操作符的“个数”是语法错误。

8.6 注意复制构造函数应使用引用调用,而非传值调用,否则复制构造函数调用会造成无穷递归(这是个致命逻辑错误),因为对于传值调用,建立传入复制构造函数的对象副本会造成复制构造函数的递归调用。

8.7 如果构造函数仅将源对象的指针复制到目标对象的指针,这两个对象将指向同一块动态分配的内存块,执行析构函数时将释放该内存块,导致另一个对象的 ptr 未被定义,这种情况可能会引起运行时错误。

8.8 类的对象包含的指向动态分配的内存的指针,但如果不为其提供重载的赋值操作符和复制的构造函数会造成逻辑错误。

## 良好编程习惯

8.1 针对同样的操作,使用重载操作符比使用显式函数调用更能提高程序的可读性。

- 8.2 由于过多或前后不一致地使用操作符重载会使程序晦涩难懂,所以应尽量避免。
- 8.3 重载操作符用于类的对象时,其功能类似于该操作符作用于内部类型的对象时完成的功能,避免无目的地滥用重载操作符。
- 8.4 用重载操作符编写C++程序之前,查阅编辑器手册,了解特定操作符的各种限制和要求。
- 8.5 要保证相关操作符的一致性,可以用一个操作符实现另一个操作符(例如,用重载的操作符“+”实现重载的操作符“+=”)。
- 8.6 重载一元操作符时,把操作符函数用作类的成员而非友元函数。因为友元函数的使用破坏了类的封装,所以除非绝对必要,否则应尽量避免使用友元函数和友元类。

### 性能提示

- 8.1 可以把一个操作符作为一个非成员、非友元函数重载。但是,这样的操作符函数访问类的 private 和 protected 数据时必须使用类的 public 接口中提供的 set 或 get 函数(即设置数据和读取数据的函数),调用这些函数的开销会降低性能,因此必须将这些函数内联以提高性能。
- 8.2 与先执行隐式类型转换然后再执行连接操作相比,使重载的连接操作符+=只有一个常量 char \* 类型参数,执行效率更高。隐式类型转换需要的代码较少,出错也较少。

### 软件工程知识

- 8.1 操作符重载提供了C++的可扩展性,这也是C++语言最吸引人的属性之一。
- 8.2 操作符函数的参数至少有一个必须是类对象或对类对象的引用。这样可防止程序员改变操作符对内部类型的对象的作用方式。
- 8.3 不修改 ostream 类和 istream 类的声明或 private 数据成员,也可为用户自定义类型添加新的输入/输出功能。这又一次证明了C++编程语言的可扩展性。
- 8.4 通常应把构造函数、析构函数、重载的赋值操作符以及复制构造函数一起提供给使用动态内存分配的类。
- 8.5 防止一个类对象被复制是能够实现的。具体作法是将赋值操作符声明为该对象的 private 成员。
- 8.6 防止类对象被复制是能够实现的;只须令重载的赋值操作符和复制构造函数为 private 即可。
- 8.7 使用转换构造函数实现隐式转换时,C++只会使用一个隐式的构造函数调用来满足重载赋值操作符的需要。通过执行一系列隐式的、用户自定义的类型转换来满足重载操作符的需要是不可能的。
- 8.8 通过以前面定义的成员函数实现成员函数,程序员可重用代码,减少了编写代码的工作量。

### 测试和调试提示

- 8.1 从 String 类的重载下标操作符返回 char 引用是危险的。例如,客户可以用这个引用在

字符串中任何位置插入空中止符( '\0' )。

### 自测题

#### 8.1 填空题:

- 假设 a 和 b 是两个整型变量,我们用 a + b 的形式求这两个变量的和;假设 c 和 d 为浮点型变量,我们用 c + d 的形式求这两个变量的和。显然,操作符 + 具有不同的用途,这是\_\_\_\_\_的例子。
- 关键字\_\_\_\_\_引入了重载操作符函数的定义。
- 要对类对象使用操作符,除了操作符\_\_\_\_\_和\_\_\_\_\_以外,其他的都必须重载。
- 重载不能改变操作符的\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

#### 8.2 解释C++ 中操作符 << 和 >> 的多重含义。

#### 8.3 C++ 中,何时可用 operator/名称?

#### 8.4 (判断正误)在C++ 中,只能重载现有的操作符。

#### 8.5 C++ 中,重载操作符的优先级和原先未重载的操作符的优先级相比,哪一个优先级更高?

### 自测题答案

#### 8.1 a) 操作符重载

b) operator

c) =, &

d) 优先级、结合性和个数

#### 8.2 根据使用场景,操作符 >> 可能是右移位操作符,也可能是流读取操作符。同样,操作符 << 可能是左移位操作符,也可能是流插入操作符。

#### 8.3 操作符重载时可以使用 operator/,它可以是提供操作符/的重载版本的函数名。

#### 8.4 正确。

#### 8.5 相同。

### 练习题

#### 8.6 尽量列举C++ 中隐式操作符重载的例子。列举需要在C++ 中显式重载操作符的典型示例。

#### 8.7 C++ 中不能重载的操作符有\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

#### 8.8 字符串连接需要两个操作数,即两个要被连接的字符串。本章介绍了如何实现将第二个 String 对象连接到第一个 String 对象右边的一个重载的连接操作符,这种连接会修改第一个 String 对象。有些实际应用中,需要在不修改 String 参数的情况下产生一个已连接的 String 对象,实现允许如下操作的操作符

```
string1 = string2 + string3;
```

#### 8.9 (基本操作符重载练习)列出C++ 所有可重载的操作符,针对每个可重载的操作符,列出它们用于几个不同类时的一种或者几种可能的用法。建议练习下面的类:



- a) 数组;
- b) 堆栈;
- c) 字符串。

完成后,说明哪些操作符的用法可适用于大量的类,哪些操作符重载价值极小,哪些操作符具有歧义性。

- 8.10 现在将上一个描述过程反过来,列出C++中每个可重载的操作符,对于每个操作符,列出你认为它应该代表的基本操作。如果有非常好的操作,把它们列出来。
- 8.11 (工程)C++是发展中的编程语言之一,许多新的语言也不断出现除现有的操作符外,还有哪些操作符可以添加到C++或添加到C++这样既支持过程化编程又支持面向对象编程的未来语言中? 将你的建议寄给ANSI C++委员会或新闻组 comp. std. C++。
- 8.12 重载函数调用操作符( )的一个范例是允许使用更常见的二维数组下标。对数组对象而言,以下表示方法

```
chessBoard[row][column]
```

可改为另一种常用的表示方法以重载函数调用操作符

```
chessBoard(row,column)
```

- 8.13 生成 DoubleSubscriptedArray 类,与图 8.4 中 Array 类的特性相似。构建时,类应生成任意行数和列数的数组。类用 operator( )进行双下标操作。例如,在 3×5 的 DoubleSubscriptedArray 数组 a 中,用户可以用 a(1,3)访问行 1 列 3 的元素。记住,operator( )可以接收任何参数(关于 operator( )的例子,参见图 8.5 的 String 类)。双下标数组的基本表达方式 rows \* columns 个元素的单下标数组。函数 operator( )应通过正确的指针算法访问数组的每个元素。实际上,operator( )应有两个版本,一个返回 int &,使 DoubleSubscriptedArray 的元素可以用作左值,一个返回 const int &,使 const DoubleSubscriptedArray 的元素可以用作右值。这个类还提供下列操作符: ==、!=、=、<< (以行和列格式输出数组)和 >> (输入整个数组内容)。
- 8.14 重载下标操作符使之返回集合中最大元素、第二大元素以及第 3 大元素等。
- 8.15 研究图 8.7 中的 Complex 类,该类可以对所谓的复数执行操作,复数的格式为

```
realPart imaginaryPart * i
```

其中 i 的值为

$$\sqrt{-1}$$

- a) 修改该类,使之能用重载的 >> 和 << 输入和输出复数(当然要从 Complex 类中删除 print 函数)。
- b) 重载乘法操作符,使之能执行两个复数的代数乘法。
- c) 重载操作符 == 和 !=,使之能比较两个复数。

```
1 //Fig. 8.7: complex1.h
2 //Definition of class Complex
3 #ifndef COMPLEX1_H
4 #define COMPLEX1_H
5
6 class Complex {
7 public:
```

```

8   Complex( double = 0.0, double = 0.0 );           //constructor
9   Complex operator +( const Complex & ) const;      //addition
10  Complex operator -( const Complex & ) const;      //subtraction
11  const Complex &operator =( const Complex & );    //assignment
12  void print() const;                               //output
13 private;
14  double real;           //real part
15  double imaginary;      //imaginary part
16  |;
17 18 #endif

```

图 8.7 complex 类——complex1.h

```

19 //Fig. 8.7: complex1.cpp
20 //Member function definitions for class Complex
21 #include <iostream>
22
23 using std::cout;
24
25 #include "complex1.h"
26
27 //Constructor
28 Complex::Complex( double r, double i )
29   : real( r ), imaginary( i ) {}
30
31 //Overloaded addition operator
32 Complex Complex::operator +( const Complex &operand2 ) const
33 {
34     return Complex( real + operand2.real,
35                     imaginary + operand2.imaginary );
36 }
37
38 //Overloaded subtraction operator
39 Complex Complex::operator -( const Complex &operand2 ) const
40 {
41     return Complex( real - operand2.real,
42                     imaginary - operand2.imaginary );
43 }
44
45 //Overloaded = operator
46 const Complex& Complex::operator =( const Complex &right )
47 {
48     real = right.real;
49     imaginary = right.imaginary;
50     return *this; //enables cascading
51 }
52
53 //Display a Complex object in the form: (a, b)
54 void Complex::print() const
55 { cout << '(' << real << ", " << imaginary << ')'; }

```

图 8.7 complex 类——complex1.h

```
56 //Fig. 8.7: fig08_07.cpp
57 //Driver for class Complex
58 #include <iostream>
59
60 using std::cout;
61 using std::endl;
62
63 #include "complex1.h"
64
65 int main()
66 {
67     Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
68
69     cout << "x: ";
70     x.print();
71     cout << "ny: ";
72     y.print();
73     cout << "nz: ";
74     z.print();
75
76     x = y + z;
77     cout << " \n\nx = y + z; \n";
78     x.print();
79     cout << " = ";
80     y.print();
81     cout << " + ";
82     z.print();
83
84     x = y - z;
85     cout << " \n\nx = y - z; \n";
86     x.print();
87     cout << " = ";
88     y.print();
89     cout << " - ";
90     z.print();
91     cout << endl;
92
93     return 0;
94 }
```

输出结果:

x: (0, 0)

y: (4.3, 8.2)

z: (3.3, 1.1)

x = y + z;

(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z;

(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)

图 8.7 complex 类——fig08\_07.cpp

8.16 32 位整数的机器所能表示的整数范围大致是 -20 亿 ~ +20 亿, 这个范围内的操作一般不会出现问題。但是有很多应用程序可能要使用超出上述范围的整数, C++ 可以满足这个需求, 但需要建立一个新的数据类型。研究图 8.8 中的 HugeInt 类, 然后完成下列各题:

- a) 准确描述它是如何操作的。
- b) 该类有什么限制?
- c) 重载乘法操作符 \*。
- d) 重载除法操作符/。
- e) 重载所有的关系操作符和相等操作符。

```

1 //Fig. 8.8: hugeintl.h
2 //Definition for class HugeInt
3 #ifndef HUGEINT1_H
4 #define HUGEINT1_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 class HugeInt {
11     friend ostream &operator <<( ostream &, const HugeInt & );
12 public:
13     HugeInt( long = 0 );           //conversion/default constructor
14     HugeInt( const char * );      //conversion constructor
15     HugeInt operator +( const HugeInt & ); //add another HugeInt
16     HugeInt operator +( int );    //add an int
17     HugeInt operator +( const char * );//add an int in a char *
18 private:
19     short integer[ 30 ];
20 };
21
22 #endif

```

图 8.8 巨型整数类——hugeintl.h

```

23 //Fig. 8.8: hugeintl.cpp
24 //Member and friend function definitions for class HugeInt
25 #include <cstring>
26 #include "hugeintl.h"
27
28 //Conversion constructor
29 HugeInt::HugeInt( long val )
30 |
31     int i;
32
33     for ( i = 0; i <= 29; i ++ )
34         integer[ i ] = 0; //initialize array to zero
35
36     for ( i = 29; val != 0 && i >= 0; i -- ) |

```

```
37     integer[ i ] = val % 10;
38     val /= 10;
39 }
40 }
41
42 HugeInt::HugeInt( const char *string )
43 {
44     int i, j;
45
46     for ( i = 0; i <= 29; i ++ )
47         integer[ i ] = 0;
48
49     for ( i = 30 - strlen( string ), j = 0; i <= 29; i ++, j ++ )
50         if ( isdigit( string[ j ] ) )
51             integer[ i ] = string[ j ] - '0';
52 }
53
54 //Addition
55 HugeInt HugeInt::operator +( const HugeInt &op2 )
56 {
57     HugeInt temp;
58     int carry = 0;
59
60     for ( int i = 29; i >= 0; i -- ) {
61         temp.integer[ i ] = integer[ i ] +
62             op2.integer[ i ] + carry;
63
64         if ( temp.integer[ i ] > 9 ) {
65             temp.integer[ i ] %= 10;
66             carry = 1;
67         }
68         else
69             carry = 0;
70     }
71
72     return temp;
73 }
74
75 //Addition
76 HugeInt HugeInt::operator +( int op2 )
77 { return *this + HugeInt( op2 ); }
78
79 //Addition
80 HugeInt HugeInt::operator +( const char *op2 )
81 { return *this + HugeInt( op2 ); }
82
83 ostream& operator <<( ostream &output, const HugeInt &num )
84 {
85     int i;
86
87     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i ++ )
```





# 第9章 继 承

## 学习目标

- 能通过继承从现有的类建立新类
- 理解继承如何提高软件的可重用性
- 理解基类和派生类的概念
- 能用多重继承从多个基类派生新类

## 9.1 简介<sup>①</sup>

随后两章将讨论面向对象编程最重要的两个特性——“继承”和“多态性”。继承是软件可重用性的一种形式,新类通过继承这一方式,从现有的类中吸收其属性和行为,并对其覆盖或改写,产生新类所需要的功能。软件的可重用性可节省程序开发的时间。它鼓励人们重复使用已经得到认可、并通过调试的高质量软件,系统运行后发现问题的可能性大为减少。这是非常振奋人心的。多态性可以使我们以常规方式写程序以操纵多种现有的、且已专门化了的相关类。继承和多态性是管理软件复杂性的有效技术。

创建新类时,程序员无须从头编写数据成员和成员函数,只须指明“新类”所要“继承”的已定义的“基类”的数据成员和成员函数。类似的新类就叫派生类。每个派生类本身还可以是其未来的派生类的基类。在简单继承中,一个基类派生一个类。而多重继承中,一个类可以从多个(彼此间可能无关联)基类中派生。简单继承很容易理解,几个简单的例子即可让你快速掌握。多重继承较为复杂、易出错,我们仅介绍一个例子,同时强烈建议你小心使用,在使用其强大功能之前一定要进行透彻的研究。

派生类也可以增加自己的数据成员和成员函数,因此派生类比基类更大。派生类由于代表更小的一组对象,因此比基类更具体。利用简单继承派生出来的派生类开始与基类的本质是一样的。继承真正优越性来自于在派生类中定义对基类特征的追加,替代和精简。

C++ 提供了 3 类继承:public,protected 和 private。本章将集中介绍 public 继承,同时简单介绍其他两类继承。第 15 章将介绍 private 继承如何被用作组合的另一种形式。第 3 类即 protected 继承在 C++ 中出现的相对较晚,用得也比较少。public 继承产生的派生类,它的每个对象也可被认为是这个派生类对应的基类的一个对象。反之则不然,基类对象不是该基类的派生类的对象。我们将利用这种“派生类对象是基类对象”的关系来做一些有趣的操作。例如可利用继承将大量不同又却相关的对象串联成一系列相关联的基类对象。这样,大量的对象就可以用统一的方法来处理。在下一章,我们将看到这种被称为多态性的技

---

<sup>①</sup> 本章和第 10 章介绍的许多方法将随 C++ 组织逐渐着手改变 C++ 标准中指定的新方法发生改变。我们将在第 21 章讨论这些新方法,如运行时类型信息(RTTI)。



术,它是面向对象编程中的关键。

本章还增加了一种新的成员访问控制形式,叫做 `protected` 访问。派生类及其友元可以访问 `protected` 基类成员,非友元函数、非派生类成员函数则不能。

构造软件系统的经验表明:很大一部分代码用于处理类似的相关具体案例。在这样的系统中很难看到“出色的代码”,因为设计者和程序员专注于具体案例。面向对象编程就提供了许多“透过现象看本质”的方法,其中一个便是“抽象”。

对于一个充满了相似相关具体案例的程序,常常可以看到用 `switch` 语句来区分具体案例并逐个提供处理这些案例的处理逻辑。第 10 章将介绍如何利用继承和多态性用简单逻辑来替代类似的 `switch` 逻辑。

我们将“是一”的关系和“有一”的关系区分开来。“是一”的关系就是继承。在这种“是一”的关系,派生类的一个对象可以视为基类的一个对象。“有一”的关系是一个合成,在这种关系里,该类的对象有一个或多个对象作为它的成员。

派生类不能访问其基类的 `private` 成员,因为这样做违反了基类的封装性。然而派生类可以访问它基类的 `public` 和 `protected` 成员。继承时,基类中不想被派生类访问的成员被定义为 `private`。派生类只有通过基类提供的 `public` 和 `protected` 函数来访问这些 `private` 成员。

继承带来一个问题是,派生类继承 `public` 成员函数其实现不需要也不应该有明确的声明。当一个基类的成员不适合该派生类时,可以在派生类中以适合的方式重载它。在某些情况下,则不适合 `public` 继承。

最令人激动的可能是新类可以从现有的类库中继承而来。有专门的组织利用在世界范围内都可用的类库来开发自己的类库。最终,软件将基本上可以由标准重用组件组合而成,如同现在的硬件一样。这对于开发未来所需的、更强大的软件有着非常大的意义。

## 9.2 继承:基类与派生类

通常情况下,一个类的对象其实“也是一个”另一个类的对象。矩形实际上也是四边形(正方形、平行四边形和梯形也是四边形)。因此, `Rectangle` 类可以说是从 `Quadrilateral` 类继承来的。`Quadrilateral` 类是基类, `Rectangle` 是派生类。矩形是四边形的特殊类型,但四边形不一定是矩形(例如四边形也可能是平行四边形)。图 9.1 列举了部分简单的继承示例。

| 基类       | 派生类                  |
|----------|----------------------|
| Student  | GraduateStudent      |
|          | UndergraduateStudent |
| Shape    | Circle               |
|          | Triangle             |
|          | Rectangle            |
|          | CarLoan              |
| Loan     | HomeImprovementLoan  |
|          | MortgageLoan         |
|          | FacultyMember        |
| Employee | StaffMember          |
|          | CheckingAccount      |
| Account  | SavingsAccount       |

图 9.1 简单的继承示例

不同的面向对象的程序语言如 SmallTalk 和 Java 使用的术语也有所不同。在继承方面,基类可能被称为超级类(superclass),即对象的父集,派生类叫子类(subclass),即对象的子集。因为继承往往令派生类拥有比基类更多的特性,父集和子集的说法容易造成误解,所以这里我们避免用这样的术语。但派生类的对象可以被认为是基类的对象,从这个意义上说,基类拥有的对象比派生类要多,因此称基类为父类,派生类为子类也是有道理的。

继承构成了一个树状的层次结构。基类和派生类间存在着层次关系。一个类可以独立存在,但一旦使用了继承的机制,这个类不是供给其他类属性和行为的基类,就是继承基类属性和行为的派生类。

以下列简单的继承层次结构为例。一所普通大学社区有数以千计的社区成员。这些成员由职工、在校生和毕业生组成。职工的身份不是教员,就是后勤人员。教员不是行政管理人员(如院长和系主任),就是教师。这便构成了如图 9.2 所示的继承层次图。注意,许多行政管理人员同时也是教师,因此使用多重继承就构成 AdministratorTeacher 类。同理,因为学生常为学校服务,职工通常教授多门课程,也可用多重继承来创建一个叫 EmployeeStudent 的类。

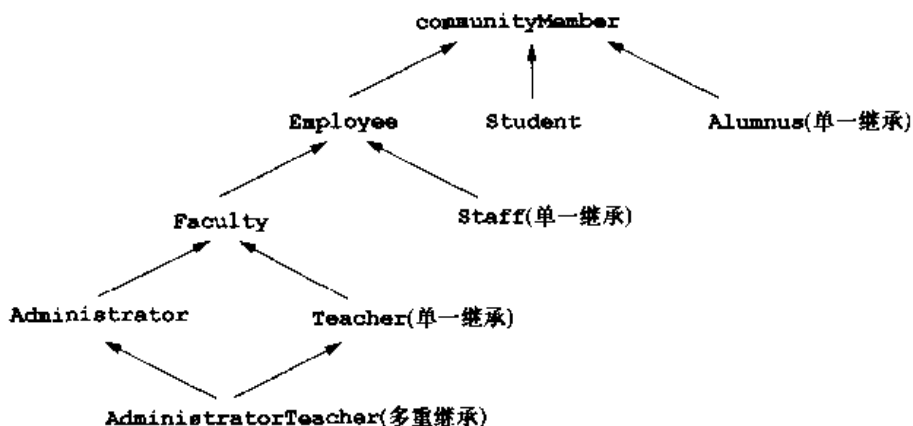


图 9.2 大学社区成员的继承层次结构

另一个真实的例子,是一个 Shape 类的层次图,如图 9.3 所示。学习面向对象编程的同学们很容易在现实世界里找到大量的类似的层次实例。只是他们还不习惯以这种方式对现实世界进行分类,因此有必要调整思维方式。

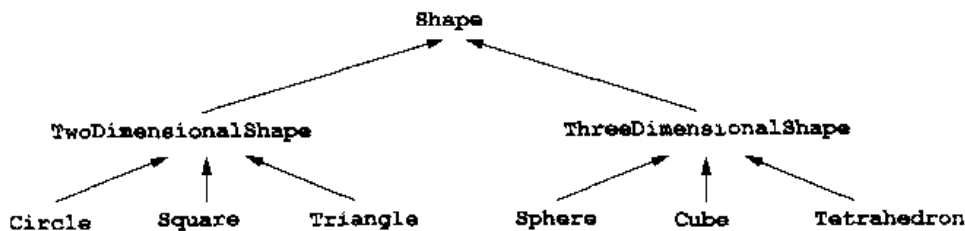


图 9.3 Shape 类的继承层次结构

下面考察一下说明继承的语法。CommisionWorker 类从 Employee 类派生而来,其定义格式一般如下

```
class CommissionWorker:public Employee|  
...  
|;
```

这叫做 public 继承,也是继承中最常见的。将来我们还会讨论 private 继承和 protected 继承。public 继承,基类里的 public 和 protected 成员分别被继承为派生类的 public 和 protected 成员。记住,基类的 private 成员在派生类中是不能被访问的。注意,友元函数也不能被继承。

完全可以把基类对象和派生类对象看成是相似的。而这些共同点就表现在基类的属性和行为中。任何从同一个基类通过 public 继承派生出的类其对象都可以看作是这个基类的对象。在以后我们会列举许多利用这种关系来简化编程的例子,非面向对象的语言(如 C)是无法做到这一点的。

### 9.3 protected 成员

基类的 public 成员可以允许程序中的所有函数访问。而基类的 private 成员只能允许基类的成员函数和友元函数访问。

我们引入 protected 访问作为 public 访问和 private 访问的中间保护层。基类的 protected 成员仅可以被基类的成员和友元函数及派生类的成员和友元函数来访问。

派生类的成员引用基类的 public 和 protected 成员时只须引用其名称。注意,protected 数据可能会打破封装——任何对基类 protected 成员的修改都可能导致对所有派生类的修改。

**软件工程知识 9.1** 通常情况下,除非系统为迎合特殊的性能要求而需要调整,否则不要将类的数据成员声明为 private 和使用 protected。

### 9.4 基类指针向派生类指针的强制类型转换

public 派生类对象也可以视为其对应的基类对象。这样就可以执行许多有趣的操作。例如,尽管从同一个基类派生了许多派生类,他们的对象不尽相同,但只要我们把这些对象视为其基类的对象,仍可以将它们创建成一个链表。反之则不然:基类的对象不能自动强制转换为派生类对象。

**常见编程错误 9.1** 将基类对象视为派生类对象会导致错误。

然而程序员常显式将基类指针强制转变为派生类指针。这一过程常称为向下强制转换指针类型。值得注意的是,要解除这样的指针参照,程序员一定要确信这个指针的类型必须与其指向的对象类型相符。这里要介绍大多数编译器用于实现向下强制转换指针类型的方法,第 21 章,我们会再次介绍符合 C++ 标准最新特性的编译器中的相关内容,如运行时类型识别信息、dynamic\_cast 和 typeid。

**常见编程错误 9.2** 显式将一个指向基类对象的基类指针强制转换为一个派生类的指针,并引用基类对象中不存在的派生类的成员,会发生运行时逻辑错误。

图 9.4 所示程序中,第 1~43 行表明 Point 类的定义和 Point 成员函数的定义。第 44~106 行是 Circle 类及其成员函数的定义。第 107~147 行是驱动程序,我们将基类指针赋给派生类指针(即向上强制转换指针类型),将派生类指针赋给基类指针。

```

1  //Fig. 9.4: point.h
2  //Definition of class Point
3  #ifndef POINT_H
4  #define POINT_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 class Point {
11     friend ostream &operator <<( ostream &, const Point & );
12 public:
13     Point( int = 0, int = 0 );      //default constructor
14     void setPoint( int, int );      //set coordinates
15     int getX() const { return x; }  //get x coordinate
16     int getY() const { return y; }  //get y coordinate
17 protected:                       //accessible by derived classes
18     int x, y;                      //x and y coordinates of the Point
19 };
20
21 #endif

```

图 9.4 基类指针向派生类指针的强制转换——point.h

```

22 //Fig. 9.4: point.cpp
23 //Member functions for class Point
24 #include <iostream>
25 #include "point.h"
26
27 //Constructor for class Point
28 Point::Point( int a, int b ) { setPoint( a, b ); }
29
30 //set x and y coordinates of Point
31 void Point::setPoint( int a, int b )
32 {
33     x = a;
34     y = b;
35 }
36
37 //Output Point (with overloaded stream insertion operator)
38 ostream &operator <<( ostream &output, const Point &p )
39 {
40     output << '[' << p.x << ", " << p.y << ']' ;
41
42     return output; //enables cascaded calls
43 {

```

图 9.4 基类指针向派生类指针的强制转换——point.cpp

```

44 //Fig. 9.4: circle.h
45 //Definition of class Circle
46 #ifndef CIRCLE_H
47 #define CIRCLE_H
48
49 #include <iostream>
50
51 using std::ostream;
52
53 #include <iomanip>
54
55 using std::ios;
56 using std::setiosflags;
57 using std::setprecision;
58
59 #include "point.h"
60
61 class Circle : public Point { //Circle inherits from Point
62     friend ostream &operator <<( ostream &, const Circle & );
63 public:
64     //default constructor
65     Circle( double r = 0.0, int x = 0, int y = 0 );
66
67     void setRadius( double ); //set radius
68     double getRadius() const; //return radius
69     double area() const;      //calculate area
70 protected:
71     double radius;
72 };
73
74 #endif

```

图 9.4 基类指针向派生类指针的强制转换——circle.h

```

75 //Fig. 9.4: circle.cpp
76 //Member function definitions for class Circle
77 #include "circle.h"
78
79 //Constructor for Circle calls constructor for Point
80 //with a member initializer then initializes radius.
81 Circle::Circle( double r, int a, int b )
82     : Point( a, b ) //call base-class constructor
83 { setRadius( r ); }
84
85 //set radius of Circle
86 void Circle::setRadius( double r )
87     { radius = ( r >= 0 ? r : 0 ); }
88
89 //get radius of Circle
90 double Circle::getRadius() const { return radius; }
91
92 //Calculate area of Circle

```

```

93 double Circle::area() const
94     { return 3.14159 * radius * radius; }
95
96 //Output a Circle in the form;
97 //Center = [x,y]; Radius = #.##
98 ostream& operator <<( ostream& output, const Circle& c )
99 {
100     output << "Center = " << static_cast< Point >( c )
101         << "; Radius = "
102         << setiosflags( ios::fixed | ios::showpoint )
103         << setprecision( 2 ) << c.radius;
104
105     return output;    //enables cascaded calls
106 }

```

图 9.4 基类指针向派生类指针的强制转换——circle.cpp

```

107 //Fig. 9.4: fig09_04.cpp
108 //Casting base-class pointers to derived-class pointers
109 #include <iostream>
110
111 using std::cout;
112 using std::endl;
113
114 #include <iomanip>
115
116 #include "point.h"
117 #include "circle.h"
118
119 int main()
120 {
121     Point *pointPtr = 0, p( 30, 50 );
122     Circle *circlePtr = 0, c( 2.7, 120, 89 );
123
124     cout << "Point p: " << p << "\nCircle c: " << c << '\n';
125
126     //Treat a Circle as a Point (see only the base class part)
127     pointPtr = &c;    //assign address of Circle to pointPtr
128     cout << "\nCircle c (via *pointPtr): "
129         << *pointPtr << '\n';
130
131     //Treat a Circle as a Circle (with some casting)
132     //cast base-class pointer to derived-class pointer
133     circlePtr = static_cast< Circle * >( pointPtr );
134     cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
135         << "\nArea of c (via circlePtr): "
136         << circlePtr->area() << '\n';
137
138     //DANGEROUS: Treat a Point as a Circle
139     pointPtr = &p;    //assign address of Point to pointPtr
140
141     //cast base-class pointer to derived-class pointer

```

```

142     circlePtr = static_cast< Circle * >( pointPtr );
143     cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
144         << "\nArea of object circlePtr points to: "
145         << circlePtr->area() << endl;
146     return 0;
147 }

```

输出结果:

```

Point p: [30, 50]
Circle c: Center = [120, 89]; Radius = 2.70

```

```

Circle c (via *pointPtr): [120, 89]

```

```

Circle c (via *circlePtr);
Center = [120, 89]; Radius = 2.70
Area of c (via circlePtr): 22.90

```

```

Point p (via *circlePtr);
center = [30, 50]; Radius = 0.00
Area of object circlePtr points to: 0.00

```

图 9.4 基类指针向派生类指针的强制转换——fig09\_04.cpp

先来考察一下 Point 类的定义。public 的 Point 类包括了 setPoint, getX 和 getY 成员函数。Point 的数据成员 x 和 y 指定为 protected, Point 对象的使用者不能直接访问它的数据, 而 Point 的派生类却可以直接访问继承来的数据成员。如果数据成员是 private, 则一定要使用 Point 的 public 成员函数来访问这些数据成员, 即使是 Point 的派生类。注意 Point 的字符流输出符重载函数可以直接访问变量 x 和 y, 因为它们是 Point 类的友元。但在引用时必须通过对象引用 x 和 y 来完成, 如 p.x 和 p.y。这是因为重载后的字符流输出符函数不是 Point 类的成员函数, 因此我们必须用显性处理以便编译器可以识别我们引用的是哪个对象。Point 类还提供了内嵌的 public 成员函数 getX 和 getY, 这样一来, 即使操作符 << 不是友元同样可获得良好的性能。但是并非所有类都会提供所需的 public 成员函数, 因此需要经常使用友元。

Circle 类是从 Point 类以 public 方式继承而来。第 1 行类定义(第 63 行)

```

Class Circle; public Point { //Circle inherits from Point

```

类定义开头的冒号表明这是继承生成。关键字 public 表明继承的类型。9.7 节将讨论 protected 和 private 继承。Point 类的所有 public 和 protected 成员在 Circle 类中被相应的被继承为 public 和 protected 成员。也就是说, Circle 的 public 成员包括了 Point 的 public 成员, 和 Circle 本身的 public 成员 area, setRadius 和 getRadius。

Circle 构造函数必须先调用 Point 的构造函数以初始化 Circle 对象的 Point 基类部分。成员的初始化(参见第 7 章)代码

```

Circle::Circle(double r, int a, int b)
:Point(a, b) //call base-class constructor

```

构造函数的前 2 行传名调用 Point 的构造函数。变量 a 和 b 由 Circle 的构造函数传给 Point 的构造函数以初始化基类成员 x 和 y。如果 Circle 的构造函数没有显式调用 Point 构造函

数,就会调用默认的 Point 构造函数,同时将默认值传给 x 和 y(如 0 和 0)。如本例中的 Point 类没有指定默认的构造函数,编译器就会报告语法错误。注意 Circle 重载后的操作符 << 函数通过将 Circle 类的引用 C 强制转换成 Point 类,来输出 Circle 的 Point 部分。这样,调用 Point 的操作符 << 并按照 Point 的格式将 x 和 y 坐标输出。

驱动程序中创建了 PointPtr 指针指向一个 Point 对象并实例化 Point 对象为 P,又创建 Circle 指针指向 Circle 对象,并实例化 Circle 对象为 C。对象 P 和 C 用重载的字符输出符来输出来显示他们是否被正确的初始化。紧接着,将派生类指针(对象 C 的地址)赋值给基类指针 PointPtr,并用 Point 的操作符 << 输出 Circle 对象 C 和被提领指针 \* PointPTR。注意:只有 Circle 对象 C 的 Point 部分显示出来了。用 public 继承,完全可以将派生类指针赋给基类指针,这是因为派生类对象就是基类对象。基类指针看起来只是派生类的基类部分。编译器执行了派生类指针向基类指针的隐式转换。

这段程序同时还说明了 PointPtr 向 Circle 的强制类型转换。这一强制转换操作的结果将赋给 CirclePtr。使用 Circle 的重载流插入操作符和提领指针 \* CirclePtr 后,输出结果是 Circle 对象 C。Circle 对象 C 的面积即是 CirclePtr 的输出结果。这同样可得出有效的面积值,因为指针始终指向 Circle 对象。

基类指针不能直接赋值给派生类指针,因为这本身是个危险赋值操作——派生类指针指向派生类对象。这种情况下编译器不会执行一个隐式转换。使用显式转换告诉编译器,程序员知道这种指针类型转换很危险——程序员有责任适当使用指针,因此编译器将会默认接受这一危险的转换。

接下来,程序把基类指针(对象 P)赋给基类指针 PointPtr,并将 PointPtr 强制转换为 Circle。强制转换的结果赋给 CirclePtr。Point 对象 P 是使用 Circle 操作符 << 和提领指针 \* CirclePtr 的输出结果。Radius 成员的输出值是 0(实际上 CirclePtr 是指向 Point 对象,而这样的成员不存在于 Point 对象中)。把一个 Point 对象输出为一个 Circle 类,结果 Radius 得到一个未定义的值(本例碰巧是 0)。Point 对象并没有类似 Radius 的成员,因此程序将此时在内存中 CirclePtr 期望 Radius 数据成员所在的位置上的值输出。用 CirclePtr 输出的是 CirclePtr(Point 对象 P)所指的对象的面积。注意这个面积值是 0.00,因为计算是基于为定义的 Radius 值。很显然,访问不存在的数据成员不会有危险,但调用不存在的成员函数则可能使程序崩溃。

本节介绍了指针转换机制,以便为下一章继续深入介绍使用多态性面向对象编程打下坚实基础。

## 9.5 使用成员函数

派生类的成员函数可能需要访问基类的某些数据成员和成员函数。

**软件工程知识 9.2** 派生类不能直接访问其基类的 private 成员。

C++ 软件工程的一个核心部分就是这个部分。如果派生类可以访问基类的 private 成员,就会破坏基类的封装。隐藏 private 成员对于测试、查错和正确地修改系统有着重要的意义。如果派生类可以访问基类的 private 成员,那从该派生类派生而来的类也能访问这些数



据,这种对 private 数据的访问能力传递下去,会削弱类的层次结构封装的优势。

## 9.6 在派生类中改写基类成员

派生类能够改写基类的成员函数,以同样的签名给出一个新版本的函数(如果签名不同,是函数的重载而不是改写),在派生类中传名调用函数时,会自动选择派生类。派生类访问基类版本的成员函数时,可能会使用作用域分辨符。

**常见编程错误 9.3** 当基类的成员函数在派生类中被改写时,一般会在派生类版本的函数中调用基类版本,再加上些其他的功能。引用基类成员函数时,不使用作用域分辨符会导致无穷递归,这是因为派生类成员函数实际上是在调用其自身。最终会造成系统内存的大量浪费,或致命的运行时错误。

以简单的 Employee 类为例。它存储了职工的姓和名。这些信息对于 Employee 派生类中的所有职工都是相同的。现在从 Employee 派生而来的类有 HourlyWorker、PieceWorker、Boss 和 CommissionWorker。HourlyWorker 按小时发薪的,每周超过 40 小时的部分按 1.5 倍发放。PieceWorker 按计件发薪的——简单地说假设该工人仅制造了一类产品,因此 private 数据成员是生产的件数和每件的工资值。Boss 按每周固定薪资发薪。CommissionWorker 的则是每周固定的薪水标准加本周销售毛利的固定百分比计算的。为简单起见,我们只研究 Employee 类和派生类 HourlyWorker。

如图 9.5 所示,第 1~50 行,是 Employee 类的定义和 Employee 成员函数的定义。第 51~106 行是 HourlyWorker 的类定义和 HourlyWorker 成员函数定义。第 107~117 行是一段 Employee/HourlyWorker 继承层次结构的程序,用于实例化一个 HourlyWorker 对象,以及初始化并调用 Hourlyworker 成员函数 Print 以输出对象的数据。

```

1 //Fig. 9.5; employ.h
2 //Definition of class Employee
3 #ifndef EMPLOY_H
4 #define EMPLOY_H
5
6 class Employee {
7 public:
8     Employee( const char *, const char * ); //constructor
9     void print() const; //output first and last name
10    ~Employee(); //destructor
11 private:
12    char * firstName; //dynamically allocated string
13    char * lastName; //dynamically allocated string
14 };
15
16 #endif

```

图 9.5 在派生类中改写基类成员函数——employ.h

```

17 //Fig. 9.5; employ.cpp
18 //Member function definitions for class Employee

```

```

19 #include <iostream>
20
21 using std::cout;
22
23 #include <cstring>
24 #include <cassert>
25 #include "employ.h"
26
27 //Constructor dynamically allocates space for the
28 //first and last name and uses strcpy to copy
29 //the first and last names into the object.
30 Employee::Employee( const char *first, const char *last )
31 {
32     firstName = new char[ strlen( first ) + 1 ];
33     assert( firstName != 0 ); //terminate if not allocated
34     strcpy( firstName, first );
35
36     lastName = new char[ strlen( last ) + 1 ];
37     assert( lastName != 0 ); //terminate if not allocated
38     strcpy( lastName, last );
39 }
40
41 //Output employee name
42 void Employee::print() const
43     | cout << firstName << ' ' << lastName; {
44
45 //Destructor deallocates dynamically allocated memory
46 Employee::~~Employee()
47 {
48     delete [] firstName; //reclaim dynamic memory
49     delete [] lastName; //reclaim dynamic memory
50 }

```

图 9.5 在派生类中改写基类成员函数——employ.cpp

```

51 //Fig. 9.5: hourly.h
52 //Definition of class HourlyWorker
53 #ifndef HOURLY_H
54 #define HOURLY_H
55
56 #include "employ.h"
57
58 class HourlyWorker : public Employee {
59 public:
60     HourlyWorker( const char *, const char *, double, double );
61     double getPay() const; //calculate and return salary
62     void print() const; //overridden base - class print
63 private:
64     double wage; //wage per hour
65     double hours; //hours worked for week
66 };
67

```

```

68 #endif
69 //Fig. 9.5: hourly.cpp
70 //Member function definitions for class HourlyWorker
71 #include <iostream>
72
73 using std::cout;
74 using std::endl;
75
76 #include <iomanip>
77
78 using std::ios;
79 using std::setiosflags;
80 using std::setprecision;
81
82 #include "hourly.h"
83
84 //Constructor for class HourlyWorker
85 HourlyWorker::HourlyWorker( const char *first,
86                             const char *last,
87                             double initHours, double initWage )
88     : Employee( first, last )    //call base-class constructor
89 {
90     hours = initHours; //should validate
91     wage = initWage;   //should validate
92 }
93
94 //get the HourlyWorker's pay
95 double HourlyWorker::getPay() const { return wage * hours; }
96
97 //Print the HourlyWorker's name and pay
98 void HourlyWorker::print() const
99 {
100     cout << "HourlyWorker::print() is executing\n\n";
101     Employee::print(); //call base-class print function
102
103     cout << " is an hourly worker with pay of $"
104           << setiosflags( ios::fixed | ios::showpoint )
105           << setprecision( 2 ) << getPay() << endl;
106 }

```

图 9.5 在派生类中改写基类成员函数——hourly.h

```

107 //Fig. 9.5: fig.09_05.cpp
108 //Overriding a base-class member function in a
109 //derived class.
110 #include "hourly.h"
111
112 int main()
113 {
114     HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
115     h.print();
116     return 0;

```

图 9.5 在派生类中改写基类成员函数——hourly.cpp

输出结果:

```
HourlyWorker::print() is executing
Bob Smith is an hourly worker with pay of $ 400.00
```

图 9.5 在派生类中改写基类成员函数——fig09\_05.cpp

Employee 类的定义包括两个 Char \* 数据成员(FirstName 和 LastName)——和 3 个成员函数(构造函数、析构函数和 Print)。构造函数收到两个字符串,并动态分配字符数组以保存字符串。注意宏 Assert(参见第 18 章)用于决定内存是否分配给 FirstName 和 LastName。如果不能分配,程序将中断并返回错误信息,表明错误发生的条件、发生错误的行号,错误条件所在文件。再次提醒你注意,在标准 C++ 中,如果没有足够的有效内存空间,new 将抛出异常;详情参见第 13 章。因为 Employee 的数据是 private,所以,要想访问它们,只能通过成员函数 Print 输出职工的姓名。析构函数返回动态分配的内存给系统以避免“内存不足”。

HourlyWorker 类以 public 方式从 Employee 继承而来。在类定义的第一行用冒号(:)指定,如下所示

```
Class HourlyWorker :public Employee
```

HourlyWorker 可以 public 方式访问 Employee 的 print 函数和 HourlyWorker 成员函数 getpay 和 Print。注意,HourlyWorker 类定义了以 Employee::print() 为原型自己的 print 函数——这是函数改写的一个例子。因此,HourlyWorker 类可以访问两个 print 函数。HourlyWorker 类还包括了 private 数据成员 Wage 和 Hour,用于计算员工的周薪。

HourlyWorker 构造函数使用成员初始化列表语法将字符串 first 和 last 传给 Employee 构造函数,这样便初始化了基类函数以及 Hour 和 Wage 这两个成员。成员函数 getpay 用于计算 HourlyWorker 的薪水。

HourlyWorker 成员函数 print 改写了 Employee 的 print 函数。为提供更强大的功能基类成员函数常被派生类改写。有时改写的函数会调用基类版本的函数来执行一部分新任务。本例中,派生类的 print 函数调用基类的 print 函数输出职工的姓名(基类的 print 是惟一可以访问基类 private 数据成员的函数)。派生类的 print 函数还可以输出职工的薪水。注意,基类版本的 print 的调用形式

```
Employee::Print();
```

因为基类函数和派生类函数具有同样的名称和签名,基类函数必须以其类名和作用域操作符开头。否则,会调用函数的派生类版本造成无穷递归调用(如 HourlyWorker 成员函数 print 函数的自我调用)。

## 9.7 public、protected 和 private 继承

从基类派生类时,基类可能以 public、protected 或 private 方式被继承。protected 和 private 继承用得很少,使用时也需要非常小心。本书中,我们一般采用 public 继承(第 15 章将详细介绍作为另一种组合形式的 private 继承)。图 9.6 总结了各种继承方式访问派生类中基类成员的能力。第一列是基类成员类别。

从 public 基类派生时,基类的 public 成员成为派生类的 public 成员,基类的 protected 成员成为派生类的 protected 成员。基类的 private 成员永远都不能被其派生类直接访问,但可以通过基类的 public 和 protected 成员函数访问。

| 基类成员的<br>访问说明符 | 继承类继承类型                                                 |                                                         |                                                         |
|----------------|---------------------------------------------------------|---------------------------------------------------------|---------------------------------------------------------|
|                | public 继承                                               | protected 继承                                            | private 继承                                              |
| public         | 在派生类中为 public<br>任何非静态成员函数、友元函数和非成员函数都可以直接访问            | 在派生类中为 protected<br>所有的非静态成员函数和友元函数可直接访问                | 在派生类中为 private<br>所有的非静态成员函数和友元函数可直接访问                  |
| protected      | 在派生类中为 protected<br>所有非静态的成员函数和友元函数可直接访问                | 在派生类中为 protected<br>所有非静态的成员函数和友元函数可直接访问                | 在派生类中为 private<br>所有非静态的成员函数和友元函数可以直接访问                 |
| private        | 在派生类中被隐藏<br>非静态成员函数和友元函数通过基类的 public 或 protected 成员函数访问 | 在派生类中被隐藏<br>非静态成员函数和友元函数通过基类的 public 或 protected 成员函数访问 | 在派生类中被隐藏<br>非静态成员函数和友元函数通过基类的 public 或 protected 成员函数访问 |

图 9.6 派生类对基类成员的访问能力

从 protected 基类派生时,基类的 public 和 protected 成员都成为派生类的 protected 成员。从 private 基类派生时,基类的 public 和 protected 成员成为派生类的 private 成员(函数变成功能函数)。private 和 protected 继承不是“是—”关系的继承。

## 9.8 直接基类和间接基类

基类可以是派生类的直接基类,也可能是间接基类。派生类的直接基类是声明派生类时明确列在派生类头部以冒号(:)标明的。间接基类没有明确列在派生类头部,间接基类是继承的类层次中两个或更多层之上的。

## 9.9 在派生类中使用构造和析构函数

因为派生类继承了基类的成员,所以实例化派生类的对象时,必须调用基类的构造函数,用于初始化派生类对象的基类成员。基类的初始化程序(我们曾见过它使用成员初始化的语法)可以在派生类中显式调用基类的构造函数来实现,否则派生类的构造函数会调用基类的默认构造函数。

基类的构造函数和基类的赋值操作符不能被派生类继承,但派生类构造函数和赋值操作符可以调用基类的构造函数和赋值操作符。

派生类的构造函数常会调用基类的构造函数,用于初始化派生类的基类成员。如果派生类没有构造函数,派生类默认的构造函数会调用基类默认的构造函数。析构函数与构造

函数的调用正好相反,因此派生类的析构函数的调用发生在调用基类的析构函数之前。

**软件工程知识 9.3** 假设我们创建了一个派生类对象,基类和派生类都包含其他类的对象。创建派生类的对象时,首先执行的基类成员对象的构造函数,其次是基类的构造函数,然后是派生类的成员对象,最后才是派生类的构造函数。析构函数的顺序刚好相反。

**软件工程知识 9.4** 成员对象的构造顺序与其在类定义中的声明顺序一致。成员的初始化顺序会影响构造顺序。

**软件工程知识 9.5** 在继承中,基类构造函数的调用顺序是按照派生类定义中继承的先后顺序来定的。基类构造函数在派生类成员初始化程序列表出现的顺序不会影响其构造的顺序。

图 9.7 中的程序演示了基类和派生类的构造和析构调用顺序。第 1~39 行是一个简单的 Point 类,有一个构造函数、一个析构函数和 protected 数据成员 x 和 y。构造和析构函数都会打印其调用的 Point 对象。

```

1 //Fig.9.1: point2.h
2 //Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point {
7 public:
8     Point( int = 0, int = 0 ); //default constructor
9     ~Point(); //destructor
10 protected: //accessible by derived classes
11     int x, y; //x and y coordinates of Point
12 };
13
14 #endif

```

图 9.7 基类和派生类构造和析构函数调用顺序——point2. h

```

15 //Fig.9.7: point2.cpp
16 //Member function definitions for class Point
17 #include <iostream>
18
19 using std::cout;
20 using std::endl;
21
22 #include "point2.h"
23
24 //Constructor for class Point
25 Point::Point( int a, int b )
26 {
27     x = a;
28     y = b;
29
30     cout << "Point constructor: "
31         << '[' << x << ", " << y << ']' << endl;
32 }

```

```

33
34 //Destructor for class Point
35 Point::~~Point()
36 {
37     cout << "Point destructor: "
38         << '[' << x << ", " << y << '>' << endl;
39 }
40 //Fig.9.7: circle2.h
41 //Definition of class Circle
42 #ifndef CIRCLE2_H
43 #define CIRCLE2_H
44
45 #include "point2.h"
46
47 class Circle : public Point {
48 public:
49     //default constructor
50     Circle( double r = 0.0, int x = 0, int y = 0 );
51
52     ~Circle();
53 private:
54     double radius;
55 };
56
57 #endif

```

图 9.7 基类和派生类构造和析构函数调用顺序——point2. cpp

```

58 //Fig.9.7: circle2.cpp
59 //Member function definitions for class Circle
60 #include <iostream>
61
62 using std::cout;
63 using std::endl;
64
65 #include "circle2.h"
66
67 //Constructor for Circle calls constructor for Point
68 Circle::Circle( double r, int a, int b )
69     : Point( a, b ) //call base-class constructor
70 {
71     radius = r; //should validate
72     cout << "Circle constructor: radius is "
73         << radius << " [" << x << ", " << y << ']' << endl;
74 }
75
76 //Destructor for class Circle
77 Circle::~~Circle()
78 {
79     cout << "Circle destructor: radius is "
80         << radius << " [" << x << ", " << y << ']' << endl;
81 }

```

图 9.7 基类和派生类构造和析构函数调用顺序——circle2. h

```

82 //Fig. 9.7: fig09_07.cpp
83 //Demonstrate when base - class and derived - class
84 //constructors and destructors are called.
85 #include <iostream>
86
87 using std::cout;
88 using std::endl;
89
90 #include "point2.h"
91 #include "circle2.h"
92
93 int main()
94 {
95     //Show constructor and destructor calls for Point
96     |
97     Point p( 11, 22 );
98     |
99
100     cout << endl;
101     Circle circle1( 4.5, 72, 29 );
102     cout << endl;
103     Circle circle2( 10, 5, 5 );
104     cout << endl;
105     return 0;
106 }

```

图 9.7 基类和派生类构造和析构函数调用顺序——circle2. cpp

输出结果:

```

Point  constructor: [11, 12]
Point  destructor: [11, 12]

Point  constructor: [72, 29]
Circle constructor; radius is 4.5 [72, 29]

Point  constructor: [5, 5]
Circle constructor; radius is 10 [5, 5]

Circle destructor; radius is 10 [5, 5]
Point  destructor: [5, 5]
Circle destructor; radius is 4.5 [72, 29]
Point  destructor: [72, 29]

```

图 9.7 基类和派生类构造和析构函数调用顺序——fig09\_07. cpp

第 40 ~ 81 行是一个 Circle 类,它以 public 方式从 Point 类派生而来。Circle 类提供了一个构造函数、一个析构函数和一个 private 数据成员 Radius。构造函数和析构函数都会将自己调用的 Circle 对象打印出来。Circle 的构造函数调用 Point 的构造函数使用成员初始化的语法,并传递 a 和 b 的值来初始化基类的数据成员 x 和 y。

第 82 ~ 106 行是 Point/Circle 层次结构的驱动程序。首先在 main 里实例化一个 Point 的



对象。对象在创建之后马上又被删除,即先后调用了 Point 的构造函数和析构函数。随后,程序实例化 Circle 对象 Circle1。调用了 Point 的构造函数执行输出,同时传递 Circle 构造函数的值被传递,再执行 Circle 构造函数中的输出。用于实例化 Circle2,Point 和 Circle 的构造函数都被调用。注意 Circle 构造函数执行之前,执行了 Point 构造函数的主体。main 执行完毕,将调用析构函数用以释放 Circle1 和 Circle 2 的存储空间。调用 Circle 和 Point 的析构函数均可用来析构 Circle2 和 Circle1。

## 9.10 派生类向基类的隐式转换

尽管一个派生类对象也“是”基类的对象,但派生类的类型与基类的类型不同。在 public 继承方式下,派生类对象也可视为基类的对象。这之所以有意义是因为派生类中有相应的成员和基类的成员一一对应,记住,派生类的成员可能比基类更多。其他方向的赋值是不允许的,因为将基类对象赋值给派生类会导致派生类独有的成员不被定义。尽管这样的赋值不规范,但提供适当的重载赋值操作符和与/或转换的构造函数(见第 8 章),可以使之合法化。注意,本节稍后将介绍有关指针的用法同样适用于引用。

**常见编程错误 9.4** 将派生类对象赋值给相应的基类对象,然后在新的基类对象中试图引用派生类才有的成员,会导致语法错误。

以 public 方式继承,指向派生类对象的指针是可能被隐式转换为指向基类对象的指针,因为派生类对象同时也是基类对象。

用基类对象与派生类对象混合和匹配基类与派生类指针时,可采用以下 4 种方法之一:

(1)直接用基类指针指向基类对象;

(2)直接用派生类指针指向派生类对象;

(3)用基类指针指向派生类指针,这不会有危险,因为派生类对象同时也是它基类的对象。这样的代码只能指向基类成员。如果用基类的指针指向派生类才有的成员,会出现编译错误;

(4)用派生类指针指向基类对象,会导致语法错误。派生类指针必须先强制转换为基类指针。

**常见编程错误 9.5** 将基类指针强制转换为派生类指针,如果指针用于引用基类对象,而基类对象中没有派生类的对象,就会导致错误。

可以很方便地将派生类作为基类进行处理,即用基类指针操纵所有这些对象。以薪资系统为例,我们可能要按照职工花名册逐个计算周薪。这时就要用基类指针在程序中调用基类有的薪资计算规则(如果有的话)。我们需要一种方法正确为每个对象调用薪资计算规则,无论是基类或是派生类对象,都需要基类指针来完成。这种方法便是第 10 章将讨论的虚拟函数和多态性。

## 9.11 继承在软件工程中的应用

我们可以用继承定制现有软件。可以从现有的类继承属性和行为,然后增加属性和行

为(或改写基类的行为)定制类以满足我们的要求。在C++中,这样不需要派生类访问基类的源代码,派生类只须能够连上基类的对象代码。这种强大的功能对于独立软件供应商(ISV)很有吸引力。独立软件开发商们能够开发出所有类进行销售或颁发许可证,并以对象代码格式为用户提供类。用户可以从类库中迅速派生新类,无须访问独立软件开发商的源代码。独立软件开发商需要为这些目标代码提供头文件。

**软件工程知识 9.6** 理论上讲,使用者不需要看见自己继承的类的源代码。实际上,提供这些类许可使用证的厂商告诉我们,他们的客户常要求他们提供源代码。因为程序员似乎仍难把别人写的代码结合到自己的程序中。

**性能提示 9.1** 性能占全导因素时,程序员可能需要看到他们继承的类的源代码,以便能优化这些代码以满足他们的性能要求。

对于学生而言,可能难以理解大型软件项目的设计者和实施人员面临的一些问题。经历过类似项目的人几乎一致认为缩短软件开发进程的一个重要环节就是软件的可重用性。面向对象的编程普遍提倡软件重用,C++尤其如此。

通过继承切实可行的类库,软件重用的优势得以最大的发挥。随着人们对C++的普遍关注,类库也越来越受到人们的欢迎。个人电脑的普及激发了软件独立生产商开发套装软件的热情。同样也引发了类库开发与销售行业的升温。应用程序设计者利用类库开发程序,类库设计者也因为自己开发的类库而获得丰厚回报。目前,随C++编译器一起发行的类库倾向于普通应用,使用范围有限。开发应用于各个领域的类库是全世界程序员的历史使命。

**软件工程知识 9.7** 创建派生类不会影响其基类的源代码或目标代码;基类的完整性可通过继承得以保证。

基类指明了共性——基类的所有派生类都继承了基类的功能。在面向对象的设计中,设计者按照求同排异的方式设计基类。在继承基类功能的基础上,定制派生类。

**软件工程知识 9.8** 在面向对象的系统中,类与类常联系紧密。求同排异是将共有的属性和行为放在基类中,然后用继承来生成派生类。

非面向对象系统的设计者往往要尽量避免不必要的函数,同样地,面向对象系统的设计者也必须避免不必要的类。因此多余的类会加大管理难度还会降低软件的可重用性。很简单,因为重用类的用户很难在庞大的集合中确定类的位置。折衷的办法是创建较少的类、每个类提供实际的附加功能,这样的类对重用者来说过于丰富,他们可能会屏蔽多余的功能,增强类以满足自己需要。

**性能提示 9.2** 如果继承而来的类比他们需要的更大,可能会造成内存和处理资源的浪费。最好继承最能满足自己需要的类。

注意,阅读一系列派生类的声明往往使人困惑,因为被继承的成员不会出现在派生类中。事实上它们确实派生类中的。派生类文档也存在类似问题。

**软件工程知识 9.9** 派生类包含其基类的属性和行为。派生类也包含其他一些属性和行

为。使用继承,基类可能被相对派生类独立编译。只有派生类增加的属性和行为需要随基类一起编译以构造派生类。

**软件工程知识 9.10** 对基类的修改不涉及到派生类的修改,只要它们对于基类继承的 public 和 protected 方式不变。派生类可能需要重新编译。

## 9.12 合成与继承

我们已经讨论过了 public 继承支持的“是一”关系。我们还要讨论一下“有一”关系(前面的章节中可以看到例子),在这样的关系中一个类可能有其他的类作为成员——这样的关系通过组合现有的类来创建新类。例如给出 Employee, BirthDate 和 TelephoneNumber 类,说 Employee 是一个 BirthDate 或一个 Employee 是一个 TelephoneNumber 都是不对的。但可以说一个 Employee 有一个 BirthDate,一个 Employee 有一个 TelephoneNumber。

**软件工程知识 9.11** 只要成员类的 public 接口不变,对基类的修改就不涉及到对派生类的修改。但要注意,合成类可能需要重新编译。

## 9.13 “使用”关系和“知道”关系

继承和合成均鼓励在现有类的基础上建立具有较多共性的新类,以倡导软件的重用。还有其他的方法可用于使用现有的类。尽管某一个对象可能不是汽车、不包含汽车,但该对象可能使用汽车。函数可以使用这个对象,只要用指向该对象的指针、引用或对象本身调用其非 private 的成员函数即可。

一个对象可以“知道”另一个对象。知识网络中常会有这样的关系。一个对象可能包含了指向另一个指针或引用的句柄,因而可以知道另一个对象。此时就可以说一个对象和另一个对象之间具有“知道”关系,这有时也称为连接。

## 9.14 案例分析:Point, Circle 和 Cylinder 类

接下来是本章的案例分析。Point, Circle, Cylinder 构成了类的层次结构。我们先开发和使用 Point 类(参见图 9.8),我们从 Point 类派生了一个 Circle 类(参见图 9.9),从 Circle 类派生了 Cylinder 类如图 9.10 所示。

```

1 //Fig.9.8: point2.h
2 //Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 class Point |

```

```

11     friend ostream &operator <<( ostream &, const Point & );
12 public;
13     Point( int = 0, int = 0 );          //default constructor
14     void setPoint( int, int );          //set coordinates
15     int getX() const { return x; }      //get x coordinate
16     int getY() const { return y; }      //get y coordinate
17 protected;                             //accessible to derived classes
18     int x, y;                          //coordinates of the point
19 };
20
21 #endif

```

图 9.8 演示 Point 类——point2. h

```

22 //Fig. 9.8: point2.cpp
23 //Member functions for class Point
24 #include "point2.h"
25
26 //Constructor for class Point
27 Point::Point( int a, int b ) { setPoint( a, b ); }
28
29 //set the x and y coordinates
30 void Point::setPoint( int a, int b )
31 {
32     x = a;
33     y = b;
34 }
35
36 //Output the Point
37 ostream &operator <<( ostream &output, const Point &p )
38 {
39     output << '[' << p.x << ", " << p.y << ']' ;
40
41     return output;          //enables cascading
42 }

```

图 9.8 演示 Point 类——point2. cpp

```

43 //Fig. 9.8: fig09_08.cpp
44 //Driver for class Point
45 #include <iostream>
46
47 using std::cout;
48 using std::endl;
49
50 #include "point2.h"
51
52 int main()
53 {
54     Point p( 72, 115 );    //instantiate Point object p
55
56     //protected data of Point inaccessible to main
57     cout << "X coordinate is " << p.getX()

```

```

58         << "Y coordinate is " << p.getY();
59
60     p.setPoint( 10, 10 );
61     cout << "The new location of p is " << p << endl;
62
63     return 0;
64 }

```

输出结果:

```

X coordinate is 72
Y coordinate is 115

```

```
The new location of p is [10, 10]
```

图 9.8 演出 Point 类——fig09\_08.cpp

```

1 //Fig. 9.9: circle2.h
2 //Definition of class Circle
3 #ifndef CIRCLE2_H
4 #define CIRCLE2_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 #include "point2.h"
11
12 class Circle : public Point {
13     friend ostream &operator <<( ostream &, const Circle & );
14 public:
15     //default constructor
16     Circle( double r = 0.0, int x = 0, int y = 0 );
17     void setRadius( double );    //set radius
18     double getRadius() const;    //return radius
19     double area() const;         //calculate area
20 protected:                    //accessible to derived classes
21     double radius; //radius of the Circle
22 };
23
24 #endif

```

图 9.9 演示 Circle 类——circle2.h

```

25 //Fig. 9.9: circle2.cpp
26 //Member function definitions for class Circle
27 #include <iomanip>
28
29 using std::ios;
30 using std::setiosflags;
31 using std::setprecision;
32
33 #include "circle2.h"

```

```

34
35 //Constructor for Circle calls constructor for Point
36 //with a member initializer and initializes radius
37 Circle::Circle( double r, int a, int b )
38     : Point( a, b )          //call base - class constructor
39 { setRadius( r ); }
40
41 //set radius
42 void Circle::setRadius( double r )
43     { radius = ( r >= 0 ? r : 0 ); }
44
45 //get radius
46 double Circle::getRadius() const { return radius; }
47
48 //Calculate area of Circle
49 double Circle::area() const
50     { return 3.14159 * radius * radius; }
51
52 //Output a circle in the form;
53 //Center = [ x, y]; Radius = #.##
54 ostream &operator <<( ostream &output, const Circle &c )
55 {
56     output << "Center = " << static_cast< Point >( c )
57         << "; Radius = "
58         << setiosflags( ios::fixed | ios::showpoint )
59         << setprecision( 2 ) << c.radius;
60
61     return output;    // enables cascaded calls
62 }

```

图 9.9 演示 Circle 类——circle2. cpp

```

63 //Fig. 9.9: fig09_09.cpp
64 //Driver for class Circle
65 #include <iostream>
66
67 using std::cout;
68 using std::endl;
69
70 #include "point2.h"
71 #include "circle2.h"
72
73 int main()
74 {
75     Circle c( 2.5, 37, 43 );
76
77     cout << "X coordinate is " << c.getX()
78         << " \nY coordinate is " << c.getY()
79         << " \nRadius is " << c.getRadius();
80
81     c.setRadius( 4.25 );

```

```

82     c.setPoint( 2, 2 );
83     cout << "\n\nThe new location and radius of c are\n"
84           << c << "\nArea " << c.area() << '\n';
85
86     Point &pRef = c;
87     cout << "\nCircle printed as a Point is; " << pRef << endl;
88
89     return 0;
90 }

```

输出结果:

```

X coordinate is 37
Y coordinate is 43
Radius is 2.5

```

```

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area 56.74

```

```

Circle printed as a Point is; [2, 2]

```

图 9.9 演示 Circle 类——fig09\_09.cpp

```

1  //Fig. 9.10: cylindr2.h
2  //Definition of class Cylinder
3  #ifndef CYLINDR2_H
4  #define CYLINDR2_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 #include "circle2.h"
11
12 class Cylinder : public Circle {
13     friend ostream &operator <<( ostream &, const Cylinder & );
14
15 public:
16     //default constructor
17     Cylinder( double h = 0.0, double r = 0.0,
18             int x = 0, int y = 0 );
19
20     void setHeight( double );    //set height
21     double getHeight() const;    //return height
22     double area() const;         //calculate and return area
23     double volume() const;       //calculate and return volume
24
25 protected:
26     double height;               //height of the Cylinder
27 };
28

```

29 #endif

图 9.10 演示 Cylinder 类——cylinder2. h

```

30 //Fig.9.10: cylindr2.cpp
31 //Member and friend function definitions
32 //for class Cylinder.
33 #include "cylindr2.h"
34
35 //Cylinder constructor calls Circle constructor
36 Cylinder::Cylinder( double h, double r, int x, int y )
37     : Circle( r, x, y )    //call base-class constructor
38 | setHeight( h ); {
39
40 //set height of Cylinder
41 void Cylinder::setHeight( double h )
42     { height = ( h >= 0 ? h : 0 ); }
43
44 //get height of Cylinder
45 double Cylinder::getHeight() const { return height; }
46
47 //Calculate area of Cylinder (i.e., surface area)
48 double Cylinder::area() const
49 {
50     return 2 * Circle::area() +
51           2 * 3.14159 * radius * height;
52 }
53
54 //Calculate volume of Cylinder
55 double Cylinder::volume() const
56     { return Circle::area() * height; }
57
58 //Output Cylinder dimensions
59 ostream &operator <<( ostream &output, const Cylinder &c )
60 {
61     output << static_cast< Circle >( c )
62           << "; Height = " << c.height;
63
64     return output;    //enables cascaded calls
65 }

```

图 9.10 演示 Cylinder 类——cylinder2. cpp

```

66 //Fig. 9.10: fig09_10.cpp
67 //Driver for class Cylinder
68 #include <iostream>
69
70 using std::cout;
71 using std::endl;
72
73 #include "point2.h"
74 #include "circle2.h"

```



```

75 #include "cylindr2.h"
76
77 int main()
78 {
79     //create Cylinder object
80     Cylinder cyl( 5.7, 2.5, 12, 23 );
81
82     //use get functions to display the Cylinder
83     cout << "X coordinate is " << cyl.getX()
84         << "\nY coordinate is " << cyl.getY()
85         << "\nRadius is " << cyl.getRadius()
86         << "\nHeight is " << cyl.getHeight() << "\n\n";
87
88     //use set functions to change the Cylinder's attributes
89     cyl.setHeight( 10 );
90     cyl.setRadius( 4.25 );
91     cyl.setPoint( 2, 2 );
92     cout << "The new location, radius, and height of cyl are:\n"
93         << cyl << '\n';
94
95     cout << "The area of cyl is:\n"
96         << cyl.area() << '\n';
97
98     //display the Cylinder as a Point
99     Point &pRef = cyl;    //pRef "thinks" it is a Point
100    cout << "\nCylinder printed as a Point is: "
101        << pRef << "\n\n";
102
103    //display the Cylinder as a Circle
104    Circle &circleRef = cyl; //circleRef thinks it is a Circle
105    cout << "Cylinder printed as a Circle is:\n" << circleRef
106        << "\nArea: " << circleRef.area() << endl;
107
108    return 0;
109 }

```

输出结果:

```

X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

```

```

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.0
The area of cyl is:
380.53
cylinder printed as a Point is:[2, 2]

```

```

Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25

```

Area: 56.74

图 9.10 演示 Cylinder 类——fig09\_10.cpp

图 9.8 是 Point 类。第 1~42 行是 Point 类的头文件和实现文件。注意 Point 的数据成员都是 protected。所以派生 Circle 类时, Circle 的成员只能直接引用 x 和 y, 但不能访问函数。这样可优化性能。

第 43~64 行是 PointD 类的驱动程序。Main 必须用 getX 和 getY 读取 protected 数据成员 x 和 y 的值。记住, 对于 protected 数据成员, 只有其类成员和友元及其派生类的成员和友元才能访问。

下一个例子如图 9.9 所示, 它重用了图 9.8 的 Point 类及其成员函数的定义。第 1~62 行表明 Circle 类及其成员函数的定义。Circle 从 Point 类以 public 方式派生而来。也就是说, Circle 类既有 Point 成员函数, 又有自己的成员函数 setRadius, getRadius 和 area。

Circle 重载的 operator << 函数是 Circle 类的友元函数, 用于输出 Circle 的 Point 部分, 强制 Circle 的引用 C 转换为 Point, 调用 Point 的 operator << 以 Point 的格式输出 x 和 y 坐标相应的值。

驱动程序实例化 Circle 类的一个对象, 然后用 get 函数获得 Circle 对象的信息。main 既不是 Circle 的成员函数, 也不是 Circle 类的友元函数, 所以不能直接引用 Circle 类的 protected 数据。程序用 set 函数 setRadius 和 setPoint 重新设置半径。最后程序初始化变量 pRef, 它是 Point 对象类型的引用 (Point&) 并指向 Circle 对象 C。程序打印 pRef, 尽管它被初始化为一个 Circle 对象, 由于它是 Point 的对象, 所以 Circle 对象实际上作为 Point 对象进行打印输出。

最后的例子如图 9.10 所示。这里重用了 Point 类和 Circle 类及其成员函数在图 9.8 和图 9.9 中的定义。第 1~65 行是 Cylinder 类及其成员函数的定义。第 66~109 行是 Cylinder 类的驱动程序。注意, Cylinder 类从 Circle 类以 public 方式派生而来。这就是说 Cylinder 既包括了 Circle 和 Point 的成员函数, 也包括了 Cylinder 自身的成员函数 setHeight, getHeight, area (改写了 Circle 的) 和 Volume。注意, Cylinder 的构造函数需要调用它的直接基类 Circle 和间接基类 Point 的构造函数。每个派生类的构造函数仅负责调用其接上层基类的构造函数 (在多重继承条件下, 可能有多个这样的直接上层基类)。同时也应当注意, Cylinder 的重载函数 operator << 函数是一个 Cylinder 类的友元函数, 它通过强制转换 Cylinder 类的引用 C, 为 Circle 类输出 Cylinder 的 Circle 部分, 它调用 Circle 的 operator << 以 Circle 的格式输出 x、y 坐标相应的值及 Radius。

驱动程序实例化了 Cylinder 的一个对象, 使用 get 函数来获取 CylinderD 对象的信息。Main 既不是 Cylinder 的成员函数也不是 Cylinder 类的友元函数, 因此不能直接引用 Cylinder 类的 protected 数据。程序用设置函数 setHeight, setRadius 和 setPoint 重置高、半径及 Cylinder 的相应值。最后, 驱动程序将“引用 Point 对象 (Point &)”类型的引用变量 pRef, 初始化为指向 Cylinder 对象 Cyl。程序打印 pRef, 尽管它初始化为一个 Cylinder 对象, 但由于是一个 Point 对象, 因此 Cylinder 对象实际上作为 Point 对象打印输出。程序还将 Circle (Circle&) 对象引用类型的变量 CircleRef 初始化为指向 Cylinder 对象 Cyl。程序打印 pRef, 尽管它已初始化为 Cylinder 对象, 但由于它是 Circle 对象, 所以 Cylinder 对象实际上作为 Circle 对象打印输出。同时也输出了 Circle 的面积。

该例很好地说明了 public 继承方式和 protected 数据成员的定义和引用。现在大家应该更清楚地了解继承的基本概念。下一章将介绍如何在一般情况下利用多态性编程来实现继承层次结构。数据抽象、继承和多态性是面向对象编程的核心。

## 9.15 多重继承

本章至此,一直在讨论单一继承,即各个派生类从同一个基类派生而来。类可以从多个基类派生而来。这种派生叫多重继承。也就是说,一个派生类继承了多个基类的成员。这一强大功能大大改进了软件的重用性,但同时也带来了一些歧义性问题。

**良好编程习惯 9.1** 使用得当的情况下,多重继承的确有用。多重继承应用于新类型和两个或更多已有类型具有“是一”关系时(例如类型 A 是类型 B 的一个类,并且类型 A 也是类型 C 的一个类)。

以图 9.11 中的多重继承为例。Base1 类包含一个 protected 数据成员——整型的变量 Value。Base1 有一个设置 Value 值的构造函数和返回 Value 值的 public 成员函数 getData。

Base2 与 Base1 相似,只不过 Base2 的 protected 数据是字符型的变量 letter。Base2 也有一个 public 成员函数 getData,但该函数将返回字符型 letter 的值。

```

1  /Fig. 9.11: base1.h
2  //Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  class Base1 {
7  public:
8      Base1( int x ) { value = x; }
9      int getData() const { return value; }
10 protected:    //accessible to derived classes
11      int value; //inherited by derived class
12 };
13
14 #endif

```

图 9.11 演示多重继承——base1.h

```

15 //Fig. 9.11: base2.h
16 //Definition of class Base2
17 #ifndef BASE2_H
18 #define BASE2_H
19
20 class Base2 {
21 public:
22     Base2( char c ) { letter = c; }
23     char getData() const { return letter; }
24 protected:    //accessible to derived classes
25     char letter; //inherited by derived class
26 };

```

```

27
28 #endif

```

图 9.11 演示多重继承——base2.h

```

29 //Fig. 9.11; derived.h
30 //Definition of class Derived which inherits
31 //multiple base classes (Base1 and Base2).
32 #ifndef DERIVED_H
33 #define DERIVED_H
34
35 #include <iostream>
36
37 using std::ostream;
38
39 #include "base1.h"
40 #include "base2.h"
41
42 //multiple inheritance
43 class Derived : public Base1, public Base2 {
44     friend ostream &operator <<( ostream &, const Derived & );
45
46 public:
47     Derived( int, char, double );
48     double getReal() const;
49
50 private:
51     double real;    //derived class's private data
52 };
53
54 #endif

```

图 9.11 演示多重继承——derived.h

```

55 //Fig. 9.11; derived.cpp
56 //Member function definitions for class Derived
57 #include "derived.h"
58
59 //Constructor for Derived calls constructors for
60 //class Base1 and class Base2.
61 //Use member initializers to call base-class constructors
62 Derived::Derived( int i, char c, double f )
63     : Base1( i ), Base2( c ), real( f ) {}
64
65 //Return the value of real
66 double Derived::getReal() const { return real; }
67
68 //Display all the data members of Derived
69 ostream &operator <<( ostream &output, const Derived &d )
70 {
71     output << "    Integer: " << d.value
72         << "\n Character: " << d.letter

```

```

73         << "\nReal number: " << d.real;
74
75     return output;    //enables cascaded calls
76 }

```

图 9.11 演示多重继承——derived.cpp

```

77 //Fig. 9.11; fig09_11.cpp
78 //Driver for multiple inheritance example
79 #include <iostream>
80
81 using std::cout;
82 using std::endl;
83
84 #include "base1.h"
85 #include "base2.h"
86 #include "derived.h"
87
88 int main()
89 {
90     Base1 b1( 10 ), *base1Ptr = 0;    //create Base1 object
91     Base2 b2( 'Z' ), *base2Ptr = 0;    //create Base2 object
92     Derived d( 7, 'A', 3.5 );    //create Derived object
93
94     //print data members of base class objects
95     cout << "Object b1 contains integer " << b1.getData()
96         << "\nObject b2 contains character " << b2.getData()
97         << "\nObject d contains: \n" << d << "\n\n";
98
99     //print data members of derived class object
100    //scope resolution operator resolves getData ambiguity
101    cout << "Data members of Derived can be"
102        << " accessed individually:"
103        << "\n Integer: " << d.Base1::getData()
104        << "\n Character: " << d.Base2::getData()
105        << "\nReal number: " << d.getReal() << "\n\n";
106
107    cout << "Derived can be treated as an "
108        << "object of either base class: \n";
109
110    //treat Derived as a Base1 object
111    base1Ptr = &d;
112    cout << "base1Ptr ->getData() yields "
113        << base1Ptr->getData() << '\n';
114
115    //treat Derived as a Base2 object
116    base2Ptr = &d;
117    cout << "base2Ptr ->getData() yields "
118        << base2Ptr->getData() << endl;
119
120    return 0;

```

121 |

输出结果:

```

Object b1 contains integer 10
Object b2 contains character z
    Integer: 7
    Character: A
Real number: 3.5
Data members of Derived can be accessed individually:
    Integer: 7
    Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A

```

图 9.11 演示多重继承——fig09\_11.cpp

Derived 类从 Base1 类和 Base2 类多重继承而来。Derived 有 private 的双精度浮点数据成员 real 和 public 成员函数 getReal, 用于读取双精度浮点 real 的值。

注意声明多重继承非常方便, 在类 Derived 之后紧跟着一个冒号(:), 然后是用逗号分隔的基类列表。还要注意, Derived 的构造函数用成员初始化语法显式调用其基类的构造函数以构造基类 Base1 和 Base2。基类构造函数的调用按照指定的继承顺序进行, 而不是构造函数的声明顺序。如果在成员初始化程序列表中沒有显式调用基类的构造函数, 也会隐式调用其默认构造函数。

Derived 重载后的字符流插入操作符用派生类对象 d 下方的圆点(.) 打印 value、letter 和 real。这个操作符函数是 Derived 的友元函数, 所以可以直接访问 Derived 的 private 数据成员 real。同样地, 它也可以访问 Base1 和 Base2 的 protected 成员 value 和 letter。

接下来查看驱动程序的 main 函数。我们创建 Base1 类的对象 B1, 并将其初始化为整型值为 10。我们创建了 Base2 类的对象 b2, 将其初始化为字符型值为 z。还创建了 Derived 类的对象 d 将其初始化, 包括整型 7、字符型 A 和双精度浮点 3.5。

打印各个基类对象的内容可以通过调用各个对象的 getData 成员函数来完成。尽管有两个 getData 函数, 但是由于指明了引用对象 b1 版的 getData 和对象 b2 版的 getData, 所以不会混淆。

接下来用静态绑定的方式打印 Derived 类对象 d 的内容。但歧义性问题出现了, 因为这个对象包含两个 getData 函数, 一个继承 Base1, 另一个继承 Base2。使用二元域分辨符即可轻松解决这个问题, 如 d.Base1::getData() 打印整型值 Value, d.Base2::getData() 打印字符型 letter。real 的双精度浮点值打印时不会发生错误, 因为调用 d.getReal()。为了说明简单继承的“是一”的关系同样适用于多重继承, 我们将派生类对象 D 的地址赋给基类指针 Base1Ptr, 并通过 Base1Ptr 调用 Base1 的成员函数 getData 打印整型的 Value。将派生类对象 D 的地址赋给基类指针 Base2Ptr 并通过 Base2Ptr 调用 Base2 的成员函数 getData 打印字符型的 Value。这个简单示例说明了多重继承的机制, 并介绍了一个简单的歧义性的问题。多重继承是一个比较复杂的话题, 在许多高级 C++ 教材中有更详细的描述。

软件工程知识 9.12 多重继承功能强大,但也增强了系统的复杂性。设计系统时务必正确使用多重继承。能用简单继承时尽量不用多重继承。

## 9.16 【可选案例分析】对象思想:在电梯模拟程序中集成继承

现在用继承来设计电梯模拟程序,看有什么好处。前一章,我们已经将 ElevatorButton 和 FloorButton 作为两个对立的类。事实上这两个类有很多相似之处:都是按钮。要使用继承,首先要考察两者间的共性。然后提取共性建立基类 Button 和派生类 ElevatorButton 和 FloorButton。

先看看 ElevatorButton 和 FloorButton 的相似之处。图 9.12 列出了这两个类的属性和行为,两者的声明包含在图 7.24 和图 7.26 中的头文件中声明的。这两个类都有一个属性 (Pressed) 和两个行为 (PressButton 和 ResetButton)。我们把这 3 个元素放入基类 Button, ElevatorButton 和 FloorButton 从 Button 里继承这些属性和行为。在前面的实现里, ElevatorButton 和 FloorButton 分别声明了对 Elevator 类的对象的引用——Button 类也应包含该引用。

| ElevatorButton          | FloorButton             |
|-------------------------|-------------------------|
| - pressed; bool = false | - pressed; bool = false |
| + pressButton(); void   | + pressButton(); void   |
| + resetButton(); void   | + resetButton(); void   |

图 9.12 ElevatorButton 类和 FloorButton 类的属性和行为

图 9.13 中的程序对基于继承的新电梯模拟程序进行了建模。注意 Floor 类由 FloorButton 类的一个对象和 Light 类的一个对象组成, Elevator 类由 ElevatorButton 类的对象和 Door 类的对象和 Bell 类的对象组成。带空心箭头的实线是从派生类指向基类——即这条实线表示 FloorButton 类和 ElevatorButton 类是从 Button 类继承而来。

还有一个问题:派生类要改写基类的成员函数吗?如果我们比较每个类的 public 成员函数(图 7.25 和图 7.27),会发现这两个类中的 ResetButton 成员函数是一样的,不需要改写。但两者的 PressButton 函数不同。ElevatorButton 类的 PressButton 代码为

```
pressed = true;
cout << "elevator button tells elevator to prepare to leave"
    << endl;
elevatorRef.prepareToLeave( true );
```

以下则是 FloorButton 类的 PressButton 代码

```
pressed = true;
cout << "floor" << FloorNumber
    << " button summons elevator" << endl;
elevatorRef.summonElevator( FloorNumber );
```

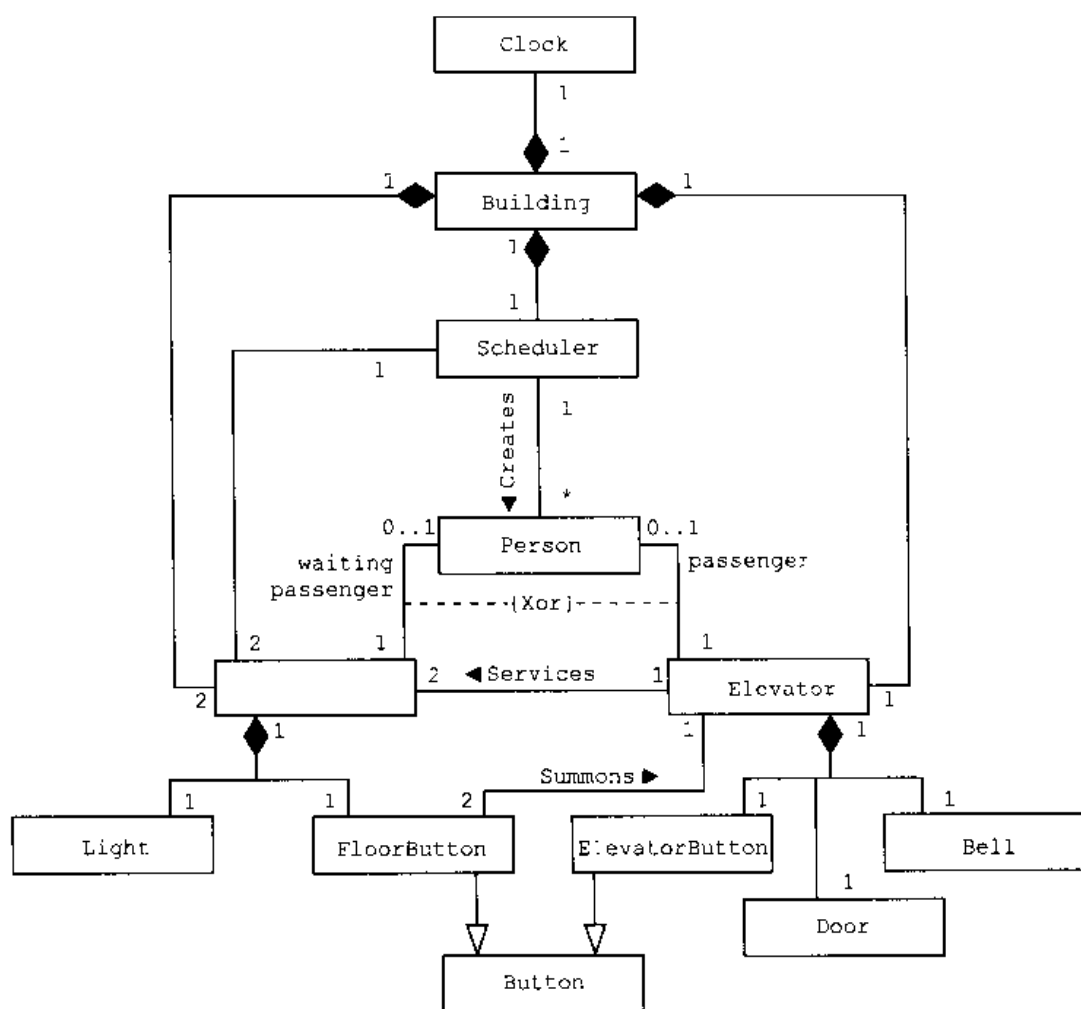


图 9.13 完整的电梯模拟程序类结构图,表明类 button 继承的情况

这两段代码的第一行相同,其余则不同。所以应在派生类里改写基类 Button 的成员函数 PressButton。

图 9.14 列出了基类 Button 的头文件<sup>①</sup>。我们声明了 public 成员函数 PressButton 和 ResetButton 及布尔值的 private 数据成员 Pressed, 注意, 第 18 行 Elevator 类的引用声明和第 11 行对应的结构函数的参数声明。我们将在讨论派生类代码的同时, 讲解如何初始化这种引用。

```

1 //button.h
2 //Definition for class Button.
3 #ifndef BUTTON_H
4 #define BUTTON_H
5
6 class Elevator;           // forward declaration

```

① 封装性的好处在于无须修改电梯模拟程序的其他源文件。我们只是用新的 elevatorButton 和 floorButton.h 和.cpp 文件代替原有的文件, 并加入新的 Button 类的文件, 然后编译新的.cpp 文件并和模拟程序的部分原有目标代码连接起来。



```

7
8 class Button |
9
10 public:
11     Button( Elevator & );           //constructor
12     ~Button();                     //destructor
13     void pressButton();             //sets button on
14     void resetButton();             //resets button off
15
16 protected:
17     //reference to button's elevator
18     Elevator &elevatorRef;
19
20 private:
21     bool pressed;                   //state of button
22 |;
23
24 #endif //BUTTON_H

```

图 9.14 Button 类的头文件

派生类执行两种不同的动作。ElevatorButton 类调用 Elevator 类的成员函数 PrepareToLeave; FloorButton 类调用 SummonElevator 成员函数。这两个类都要访问基类的 ElevatorRef 数据成员;非 Button 类的对象不能使用这类数据成员。因此我们把 ElevatorRef 数据成员放在 Button 的 protected 部分。Pressed 数据成员声明为 private,因为它只能通过基类的成员函数来操作,派生类不需要直接访问 Pressed。

图 9.15 列出了 Button 类的实现文件。第 12 行

```
    : elevatorRef(elevatorHandle),press( false )
```

初始化了电梯的一个引用。构造和析构函数只显示一些表明运行状态的信息,PressButton 和 ResetButton 成员函数则处理 private 的数据成员 Pressed。

```

1 //button.cpp
2 //Member function definitions for class Button.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "button.h"
9
10 //constructor
11 Button::Button( Elevator &elevatorHandle )
12     : elevatorRef( elevatorHandle ),pressed( false )
13 { cout << "button created" << endl; }
14
15 //destructor
16 Button::~~Button()
17 { cout << "button destroyed" << endl; }
18

```

```

19 //press the button
20 void Button::pressButton() { pressed = true; }
21
22 //reset the button
23 void Button::resetButton() { pressed = false; }

```

图 9.15 Button 类的实现文件

图 9.16 包含了 ElevatorButton 类的头文件。第 10 行通过继承从 Button 类生成了 ElevatorButton。这意味着 ElevatorButton 类包含了 protected ElevatorRef 数据成员和 public PressButton 和 ResetButton 成员函数。第 15 行提供了一个 PressButton 的函数原型来表明将我们打算改写 .cpp 文件中的成员函数。稍后将讨论 PressButton 的实现细节。

```

1 //elevatorButton.h
2 //Definition for class ElevatorButton.
3 #ifndef ELEVATORBUTTON_H
4 #define ELEVATORBUTTON_H
5
6 #include "button.h"
7
8 class Elevator;           //forward declaration
9
10 class ElevatorButton : public Button {
11
12 public:
13     ElevatorButton( Elevator & );    //constructor
14     ~ElevatorButton();              //destructor
15     void pressButton();              //press the button
16 };
17
18 #endif //ELEVATORBUTTON_H

```

图 9.16 ElevatorButton 类的头文件

构造函数取 ElevatorButton 类的引用作为参数(第 13 行)。在讨论该类的实现文件时,我们将说明这个参数的必要性。注意,仍需提前声明 Elevator 类(第 8 行),以便在构造函数的声明处包含这个参数。

图 9.17 列出了 ElevatorButton 类的实现文件。类的构造和析构函数显示的信息表明这些函数正在执行中。第 14 行

```
    :Button(ElevatorHandle)
```

将 Elevator 引用上传给基类构造函数。

```

1 //elevatorButton.cpp;
2 //Member function definitions for class ElevatorButton.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "elevatorButton.h"

```



```

21
22 #endif // FLOORBUTTON_H

```

图 9.18 FloorButton 类的头文件

图 9.19 是 FloorButton 类的实现文件。第 14 行把 Elevator 的引用传给基类 Button 的构造函数,并初始化 FloorNumber 数据成员。构造和析构函数用 FloorNumber 数据成员打印相应信息。改写后的 PressButton 成员函数(第 27~34 行)开始先调用基类的 PressButton,然后再调用 Elevator 的 SummonElevator 成员函数,并传给 FloorNumber 以表明在哪一层向电梯发出请求。

```

1 //floorButton.cpp
2 //Member function definitions for class FloorButton.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "floorButton.h"
9 #include "elevator.h"
10
11 //constructor
12 FloorButton::FloorButton( int number,
13                           Elevator &elevatorHandle )
14     : Button( elevatorHandle ), floorNumber( number )
15 {
16     cout << "floor " << floorNumber << " button created"
17         << endl;
18 }
19
20 //destructor
21 FloorButton::~~FloorButton()
22 {
23     cout << "floor " << floorNumber << " button destroyed"
24         << endl;
25 }
26
27 //press the button
28 void FloorButton::pressButton()
29 {
30     Button::pressButton();
31     cout << "floor " << floorNumber
32         << " button summons elevator" << endl;
33     elevatorRef.summonElevator( floorNumber );
34 }

```

图 9.19 FloorButton 类的实现文件

现在总算完成了自第 2 章开始的电梯模拟程序的实现。结构上还可进一步改良,你会注意 Button、Door 和 Light 类有很多共同点。每个类都有一个状态的属性和相应的开、关操作。Bell 类和其他类一样,也有相似之处。面向对象思想指出,应该把一些类的共同点放在

它们的一个或多个基类里,我们可以从这些类中通过继承建立我们想要的类。我们把这个继承的实现当成一个练习留给读者。建议从修改图 9.13 的程序开始。提示:Button、Door 和 Light 实际上都是一种“开关”类型的类—它们都有状态并可以开或关;Bell 类更小,只有一个操作,没有状态。

我们真诚地希望这个电梯模拟程序的案例分析是一次有意义、有挑战性的经历。我们采用精细的、渐进的方法设计基于 UML 的电梯模拟程序。通过这个设计,我们用 C++ 进行了实际的实现,使用了一些关键的编程标记,包括类、对象、封装、可见性、合成和继承。本书后续章节还将介绍部分关键的 C++ 技术。针对该案例分析,如有更好的建议或意见,希望你畅所欲言,发信至 [book@21bj.com](mailto:book@21bj.com)。

## 9.17 小结

- 面向对象编程的强大功能之一是可通过继承实现软件的重用。
- 编程人员可以定义一个类,这个类继承了已定义的类的成员属性和成员方法,已定义的类称为基类,新定义的类称为派生类。
- 单一继承时,一个类只有一个派生类,多重继承则是新类从多个基类(互相可能不相关)派生而来。
- 派生类一般会增加它自己的成员属性和成员方法,所以派生类的成员定义往往比基类更多。派生类比基类具体,而且通常代表更少的对象。
- 派生类不能访问其基类的 `private` 成员;这样会破坏基类的封装性,但派生类可以访问其基类的 `public` 和 `protected` 成员。
- 派生类的构造函数永远都是先调用基类的构造函数来创建和初始化它的基类成员。
- 析构函数的调用顺序和构造函数的调用顺序相反,也就是说,派生类的析构函数先于基类的析构函数调用。
- 继承使软件可以重用,这样可缩短软件开发过程,并且鼓励人们重用已经得到认可、并通过调试的高质量软件。
- 可以继承一些现有的类库。不久的将来,软件可以像硬件那样,用一些标准的可重用部件来“组装”。
- 派生类的实现并不需要其基类的源代码,但需要其基类的接口和目标代码。
- 派生类对象可以用作其相应的基类对象,但基类对象不能用作其派生类对象。
- 派生类存在于它与其直接派生类的层次关系当中。
- 类可以单独存在。但类用于继承时,不是作为为其他类提供属性和行为的基类,就是作为继承基类的属性和行为的派生类。
- 继承层次的深度是任意的,只与系统的物理限制有关。
- 继承是用于理解和处理复杂性有用工具。随着软件的日益复杂,C++ 通过继承和多态性提供支持层次结构的机制。
- 可以把基类指针显式转换为其派生类指针,基类指针不应该被转换,除非它实际上已经指向派生类的对象。

- `protected` 访问符是介于 `public` 访问符和 `private` 访问符之间。基类的 `protected` 成员可以供该基类的成员或友元和派生类的成员和友元访问,除此之外的函数不能访问基类的 `protected` 成员。
- `protected` 成员用于为派生类提供访问权限,同时拒绝那些非友元和非类函数的访问。
- 多重继承用派生类名称后的冒号(:)及随后的逗号分隔列表来表示。成员初始化列表语法用于派生类构造函数调用基类构造函数时。
- 从基类派生类的时候,基类可定义为 `public`, `protected` 或 `private`。
- 从 `public` 基类派生类时,该基类的 `public` 成员就是派生类的 `public` 成员,基类的 `protected` 成员就是派生类的 `protected` 成员。
- 从 `protected` 基类派生类时,该基类的 `public` 成员和 `protected` 成员便成为派生类的 `protected` 成员。
- 从 `private` 基类派生类时,该基类的 `public` 成员和 `protected` 成员便成为派生类的 `private` 成员。
- 基类可能是其派生类的直接或间接基类,派生类直接的基类被显式列在该派生类定义处。派生类的间接基类没有显式列出:它实际上从类继承树的多个层次之上继承而来。
- 基类成员不适合派生类时,可以很轻松地在派生类里重新定义。
- 区分“是”关系和“有”关系很重要,在一个“有”关系里,一个类的对象有一个其他类的对象的成员,在一个“是”关系里,一个派生类的对象可以当成是一个基类类型的对象,“是”是继承关系,“有”是合成关系。
- 派生类对象可以赋给基类的对象,这种赋值是有意义的,因为这个派生类的对象有基类的每一个相应的成员。
- 指向派生类对象的指针可以隐式转换为基类指针。
- 有可能把一个基类指针显式转换为派生类的指针,前提是该基类指针指向派生类对象。
- 基类指定了一些共性,从该基类派生而来的所有类继承了该基类的所有功能。在面向对象设计过程中,设计人员寻找共性并进行合成,建立符合该共性的类,派生类对基类继承而来的能力进行扩展。
- 阅读派生类的声明,可能让人云里雾里,因为并非所有成员都在这些声明里,也就是说继承而来的成员并不在派生类的声明里,实际上它们存在于派生类中。
- “有”关系是合成已有的类创建新类。
- “知道”关系是对象包含指向对象的指针或引用,所以它们可知道这些对象。
- 成员对象的构造函数是按声明顺序调用的。在继承里,基类构造函数的调用是按声明时指定的顺序并在派生类构造函数被调用之前调用的。
- 创建派生类的对象时,首先调用基类的构造函数,然后再调用派生类的构造函数(这有可能调用派生类的成员对象的构造函数)。
- 删除派生类对象时,析构函数的调用顺序和构造函数相反——首先是派生类的析构函数,其次才是基类的析构函数。

- 类可能从多个基类派生而来,这种派生叫多重继承。
- 在继承说明符后加上用逗号分隔基类的列表以表明多重继承。
- 派生类用成员初始化列表语法来逐个调用其基类的构造函数,基类按其继承时的声明顺序被调用。

## 本章术语

|                                          |                                                    |
|------------------------------------------|----------------------------------------------------|
| abstraction 抽象                           | knows a relationship “知道”关系                        |
| ambiguity in mutple inheritance 多重继承的歧义性 | member access control 成员访问控制                       |
| association 关联                           | member class 成员对象                                  |
| base class default constructor 基类的默认构造函数 | member class 成员类                                   |
| base class 基类                            | multiple inheritance 多重继承                          |
| base-class constructor 基类构造函数            | object-oriented programming (OOP)<br>面向对象编程(OOP)   |
| base-class destructor 基类析构函数             | override a base-class member function<br>改写基类的成员函数 |
| base-class initializer 基类初始化成员           | pointer to a base-class object<br>指向一个基类对象的指针      |
| base-class pointer 基类指针                  | pointer to a derived-class object<br>指向一个派生类对象的指针  |
| class hierarchy 类层次结构                    | private base class private 的基类                     |
| class libraries 类库                       | private inheritance private 继承                     |
| client of a class 类的客户代码                 | protected base class protected 基类                  |
| composition 复合                           | protected inheritance protected 继承方式               |
| customize software 定制软件                  | protected keyword protected 关键字                    |
| derived class 派生类                        | protected member of a class 类的 protected 成员        |
| derived-class constructor 派生类构造函数        | public base class public 的基类                       |
| derived-class destructor 派生类析构函数         | public inheritance public 继承方式                     |
| derived-class 派生类指针                      | single inheritance 简单继承                            |
| direct-base class 直接基类                   | software reusability 软件重用性                         |
| downcasting a pointer 向下转换一个指针           | standardized software components 标准的软件部件           |
| friend of a base class 基类的友元             | subclass 子类                                        |
| friend of a derived class 派生类的友元函数       | superclass 超类                                      |
| function overriding 函数改写                 | upcasting a point 向上转换一个指针的类型                      |
| has a relationship “有”关系                 | uses a relationship “使用”关系                         |
| hierachical relationship 层次关系            |                                                    |
| indirect base class 间接基类                 |                                                    |
| infinite rcursion error 无穷递归错误           |                                                    |
| inheritance 继承                           |                                                    |
| is a relationship “是”关系                  |                                                    |

## 常见编程错误

- 9.1 将基类对象视为派生类对象会导致错误。
- 9.2 将指向基类对象的基类指针显式强制转换为派生类指针,并引用基类对象中不存在的派生类成员,会导致运行时逻辑错误。
- 9.3 当基类的成员函数在派生类中被改写,一般会在派生类版本的函数中调用基类版本,再加上些其他功能。引用基类成员函数时,不用作用域分辨符来引用基类成员函数会导

致无穷递归,这是因为派生类成员函数实际上是在调用其自身。最终会造成系统内存的大量浪费,或致命的运行时错误。

- 9.4 将派生类对象赋值给相应的基类对象,然后在新的基类对象中试图引用派生类仅有的成员,会产生语法错误。
- 9.5 将基类指针强制转换成派生类指针,如果指针用于引用基类对象,而基类对象中没有派生类的一些对象,就会发生错误。

### 良好编程习惯

- 9.1 多重继承如果使用得当的确非常有用。多重继承应该用于新类型和两个或更多的已存在的类型间是“是一”关系的情况(例如类型 A 是一个类型 B 的类,并且类型 A 也是一个类型 C 的类)。

### 性能提示

- 9.1 性能占主导因素时,程序员可能需要看到他们继承的类的源代码,以便能优化这些代码以满足他们的性能要求。
- 9.2 如果继承而来的类比他们需要的更大,可能会造成内存和处理资源的浪费。最好继承那些最能满足自己需求的类。

### 软件工程知识

- 9.1 通常情况下,除非系统为迎合特殊的性能要求而需要调整,否则不要将一个类的数据成员声明为 `private` 和使用 `protected`。
- 9.2 派生类不能直接访问其基类的 `private` 成员。
- 9.3 假设我们创建了一个派生类对象,基类和派生类都包含了其他类的对象。创建派生类的对象时,首先执行的是基类成员对象的构造函数,其次是基类的构造函数,然后是派生类的成员对象,最后才是派生类的构造函数。析构函数的顺序刚好相反。
- 9.4 成员对象的构造顺序与其在类定义中的声明顺序一致。成员的初始化顺序会影响构造顺序。
- 9.5 在继承中,基类构造函数的调用顺序是按照在派生类定义中继承的先后顺序来定的。基类构造函数在派生类成员初始化程序列表出现的顺序不会影响其构造顺序。
- 9.6 理论上讲,使用者不需要看见自己继承的类的源代码。实际上,提供这些类许可使用证的厂商告诉我们,他们的客户常要求他们提供源代码。因为程序员们似乎仍难把别人写的代码结合到自己的程序中。
- 9.7 创建派生类不会影响其基类的源代码或目标代码;基类的完整性通过继承得以保证。
- 9.8 在面向对象的系统中,类与类常联系紧密。求同排异是将共有的属性和行为放在基类中,然后用继承来生成派生类。
- 9.9 派生类包含其基类的属性和行为。派生类也包含其他一些属性和行为。使用继承,基类可能相对于派生类独立编译。只有派生类增加的属性和行为需要随基类一起编译以构造派生类。



- 9.10 对基类的修改不涉及到派生类的修改,只要它们对于基类继承的 public 和 protected 方式不变。派生类可能需要重新编译。
- 9.11 只要成员类的 public 接口不变,对基类的修改就不涉及到对其派生类的修改。但要注意,复合类可能需要重新编译。
- 9.12 多重继承功能强大,但也增强了系统的复杂性。设计系统时务必正确使用多重继承。能用简单继承时尽量不用多重继承。

### 自测题

#### 9.1 填空题:

- a) 如果 Alpha 类是从 Beta 类继承而来,Alpha 就是\_\_\_\_\_类,Beta 类叫\_\_\_\_\_类。
- b) C++ 提供\_\_\_\_\_的机制,这样就可以让派生类从多个基类继承,即使这些基类互不相关。
- c) 继承使得\_\_\_\_\_成为可能,这样可以节省开发的时间并鼓励以前经过验证的高质量的软件。
- d) \_\_\_\_\_类的对象可以视为其相应的\_\_\_\_\_基对象。
- e) 要把基类类型的指针用作派生类类型的指针时,必须进行\_\_\_\_\_,因为编译器认为这是危险操作。
- f) 3 个成员访问说明符分别是\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- g) 以 public 方式从基类继承派生类时,基类的 public 成员即为基类的\_\_\_\_\_成员,基类的 protected 成员即为基类的\_\_\_\_\_成员。
- h) 以 protected 一个从基类继承派生一个类时,基类的 public 成员即为基类的\_\_\_\_\_成员,基类的 protected 成员即为基类的\_\_\_\_\_成员。
- i) 类与类之间的“有”的关系表示\_\_\_\_\_,类与类之间的“是”的关系表示\_\_\_\_\_。

### 自测题答案

- 9.1 a) 派生,基 b) 多重继承 c) 软件可重用 d) 派生,基 e) 强制类型转换  
f) public,protected, private g) public, protected h) protected,protected i) 合成,继承

### 练习题

- 9.2 以 Bicycle 类为例。根据你自己对自行车的了解情况,给出 Bicycle 类继承其他类(这些类可以从其他类继承而来)的层次结构图。讨论 Bicycle 类各种对象的实例化,讨论其他紧密相关的派生类对 Bicycle 类的继承性。
- 9.3 简要定义以下术语:继承、多重继承、基类和派生类。
- 9.4 讨论为什么编译器认为把基类类型的指针转换为派生类类型的指针时危险的。
- 9.5 区分简单继承和多重继承。
- 9.6 (判断正误)派生类通常又称为子类,因为它代表了其基类的一个子集。
- 9.7 (判断正误)派生类类型的对象又是其基类类型的对象。
- 9.8 一些程序员不喜欢用 protected 访问符,因为它破坏了基类的封装性。讨论用 protected 访

问符对于坚持用 `private` 访问符来访问基类成员的优点。

- 9.9 许多用继承方式编写的程序也可以用合成的方式来代替,反之亦然。讨论本章中 `Point`, `Circle` 和 `Cylinder` 类继承用这些方法来实现的优缺点。用合成方式代替继承方式来改写图 9.10 的程序(及其支持类)。然后,重新评估用这两种方式来解决 `Point`, `Circle` 和 `Cylinder` 类问题和一般的面向对象问题的优缺点。
- 9.10 把图 9.10 的 `Point`, `Circle` 和 `Cylinder` 程序为改写 `Point`, `Square` 和 `Cube` 程序。分别用两种方式:继承和合成。
- 9.11 本章指出“当基类的成员不适合于派生类时,可以在派生类里改写该成员的实现方法”,改写后,“派生类对象是基类对象的”这一关系还存在吗?请说明原因。
- 9.12 研究图 9.2 的继承层次。指出层次中每个类的共同属性和行为。在层次中增加其他类(如 `UndergraduateStudent`, `GraduateStudent`, `Freshman`, `Sophomore`, `Junior` 和 `Senior` 等)使整个层次更丰富。
- 9.13 写出 `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` 和 `Square` 类的继承层次。用 `Quadrilateral` 类作为层次的基类。尽可能把层次定义深一些。`Quadrilateral` 程序的 `private` 的数据应该是该 `Quadrilateral` 4 个顶点的  $(x, y)$  坐标值对。
- 9.14 写下所能想到的形状(二维和三维的)使其生成一个形状层次结构图。你的层次应以 `Shape` 类为基类并从它派生出 `TwoDimensionalShape` 类和 `ThreeDimensionalShape` 类。写好层次后,定义层次中的每一个类。我们将在第 10 章的练习题中里把所有形状作为基类 `Shape` 进行处理,这叫多态性。

## 第 10 章 虚拟函数和多态性

### 学习目标

- 理解多态性的概念
- 理解如何声明和利用虚拟函数来实现多态性
- 理解抽象类和具体类的区别
- 学会如何声明创建抽象类的纯虚拟函数
- 理解多态性是如何增强系统的可扩展性和可维护性
- 理解C++ 如何实现虚拟函数和动态绑定

### 10.1 简介

有了“虚拟函数”和“多态性”，设计和实现扩展性更强的系统就有了可能。程序可以对层次结构中所有现有类的对象（如基类对象）进行常规处理。程序开发时没有的类，也可以利用程序的通用部分稍作修改或不作任何修改，将其增加到系统中，只要它们是继承的一部分。程序中惟一需要修改的部分只是那些需要直接了解加入层次结构中的特定类的部分。

### 10.2 类型域和 switch 语句

对不同类型的对象进行处理，一种方法是使用 switch 语句。不同类型的对象，采用的操作有所不同。例如，在形状的层次结构中，每种形状都指定本身的类型作为数据成员，使用 switch 结构就可以根据对象的具体类型确定要调用的 print 函数。

但是，使用 switch 逻辑存在很多问题。程序员可能忘记执行必要的类测试；也可能忘记测试 switch 中所有的可能情况；如果基于 switch 的系统增加了一个新类，程序员可能会忘记将其添加到在现有的所有 switch 语句中。一个类的每次新增和删除都需要修改系统中的所有 switch 语句，但追踪这些语句非常耗时，而且容易出错。

如后文所述，使用虚拟函数和多态性进行程序设计后，将不再需要 switch 逻辑。程序员可以用虚拟函数机制自动执行等价的逻辑，从而避免了 switch 逻辑带来的各种错误。

**软件工程知识 10.1** 使用虚拟函数和多态性后，一个很有趣的结果是，程序看上去变得很简单。程序中很少有分支逻辑，而是一些简单的顺序代码。这大大简化了程序的测试、调试和维护，避免了错误。

### 10.3 虚拟函数

假设有一组形状类如 Circle, Triangle, Rectangle, Square 等等都是从基类 Shape 继承而来

的。在面向对象编程中,类似的每个类可能都可绘制自身的形状。尽管每个类都有自己的 draw 函数,但每种形状的 draw 函数不尽相同。在绘制一个形状时,无论要绘制何种形状,最好能将所有的形状都作为基类 Shape 的对象来处理。这样一来,无论绘制什么形状,所要做的工作仅是调用基类 Shape 的 draw 函数,让程序动态确定(在运行时确定)执行相应的派生类的 draw 函数。

为了实现这一目的,我们在基类中将 draw 声明为虚拟函数,然后在每个派生类中改写 draw,使之能够绘制正确的形状。声明虚拟函数时,可在基类的函数原型前加关键字 virtual。如基类 Shape 中出现的声明

```
virtual void draw() const;
```

该原型声明的 draw 函数是不带参数不返回数值的常量函数,也是个虚拟函数。

**软件工程知识 10.2** 函数一旦被声明为虚拟函数,即使类在改写它时没有将其声明为虚拟函数,它从该点之后的继承层次结构中仍然是虚拟函数。

**良好编程习惯 10.1** 尽管某个函数在类层次结构中的高层被声明为虚拟函数可使其在低层隐式成为虚拟函数,但在每层显式声明这些虚拟函数可增强程序的可读性。

**软件工程知识 10.3** 派生类不定义虚拟函数时,可简单继承其直接基类的虚拟函数定义。

如果基类中的 draw 函数已经声明为 virtual,用基类的指针或引用来指明派生类对象并用该指针(如:shapePtr->draw())或引用来调用 draw 函数,程序会根据对象的类型(而不是指针或引用的类型)动态(在运行时)选择相应的派生类的 draw 函数。这就是 10.6 和 10.9 节的案例分析中讨论的动态绑定。

如果用名称和圆点(.)成员选择操作符引用一特定对象(如:squareObject.draw())以调用虚拟函数,被调用的虚拟函数是在编译时确定的(即静态绑定),调用的虚拟函数就是为该特定对象的类(或继承该对象类)定义的函数。

## 10.4 抽象基类和具体类

把一个类视为一个数据类型时,我们假定该类型的对象要被实例化。然而,很多情况下,定义程序员不想将其实例化为任何对象的类仍然大有好处。这样的类称为“抽象类”(abstract class)。因为这些抽象类要被用作基类,所以通常也称为“抽象基类”(abstract base class)。抽象基类不能实例化为对象。

抽象类的惟一用途就是为其他类提供合适的基类,以便它们可以从它那继承和/或实现接口。可实例化为对象的类称为“具体类”(concrete class)。

例如,我们可创建一个抽象基类 TwoDimensionalShape 和从它派生而来的具体类如 Square, Circle 和 Triangle。也可创建另一个抽象基类 ThreeDimensionShape,从它派生而来的具体类如 Cube, Sphere 和 Cylinder。抽象基类代表的含义太广泛因而很难定义具体的对象。在考虑实例化对象之前,我们需要确定其含义,这就是具体类要做的工作,即提供更明确的含义使之实例化为具体的对象。

将类的虚拟函数声明为(pure)就可以将其抽象化。“纯虚拟”(pure virtual)函数是在声

明时初始化为 0 的函数。例如

```
Virtual double earnings() const =0; //pure virtual
```

**软件工程知识 10.4** 如果类从带有纯虚拟函数的类派生而来,且该派生类中没有提供该纯虚拟函数的定义,那么这个纯虚拟函数在该派生类中仍然是纯虚的,这个派生类仍然是抽象类。

**常见编程错误 10.1** 试图实例化抽象类对象(即包括一个或多个纯虚拟函数的类)会导致语法错误。

一个类的层次结构中可以不包括任何抽象类,但许多面向对象的好系统中,类层次结构的顶部便是抽象基类。某些情况下,层次结构的顶部好几层都是抽象类。形状类的层次结构就是一个典型示例。该层次结构的顶部是抽象基类 Shape。下一层有两个抽象基类 TwoDimensionalShape 和 ThreeDimensionalShape。再下一层开始定义二维形状具体类(如圆形类和方形类)和三维形状具体类(球类与立方体类)。

## 10.5 多态性

C++ 支持多态性,即通过继承相关的不同的类,对象能够对同一个函数调用做出不同响应。例如,如果 Rectangle 类从 Quadrilateral 类派生而来,Rectangle 类的对象就比 Quadrilateral 类的对象更具体,适用于 Quadrilateral 类的对象的操作(如计算周长或面积)也同样适用于 Rectangle 类的对象。

多态性是通过虚拟函数实现的。通过基类指针(或引用)来请求使用虚拟函数时,C++ 会在与对象关联的派生类中选择正确的改写过的函数。

有时,在基类中定义的非虚拟函数会在派生类中被改写。如果通过基类指针调用该成员函数,使用的将是基类版本的成员函数;如果通过派生类指针调用该成员函数,使用的则是派生类版本的成员函数。这并不属于多态性行为。

下面的例子

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print(); //call base-class print function
hPtr->print(); //call derived-class print function
ePtr = &h; //allowable implicit conversion
ePtr->print(); //still calls base-class print
```

使用了图 9.5 的基类 Employee 和派生类 HourlyWorker: 基类 Employee 和派生类 HourlyWorker 都定义了自己的 print 函数。由于它们都没有被声明为 virtual,但签名相同,所以通过 Employee 指针调用 print 函数时,调用的却是 Employee::print() (不管 Employee 指针指向基类对象还是派生类 HourlyWorker 对象),而通过 HourlyWorker 指针调用 print 函数则会调用 Worker::print()。派生类也可调用基类函数,但派生类对象通过派生类对象的指针调用基类 print 时,如下显式调用函数

```
hPtr->Employee::print(); //call base-class print function
```

它表示必须显式调用基类 print。

使用虚拟函数和多态性,可使成员函数调用会因为接收该调用的对象类型的不同而做出不同的反应(但需要少量执行时的开销)。多态性赋予程序员更强的灵活性。随后将举例说明多态性和虚拟函数的功能。

**软件工程知识 10.5** 利用虚拟函数和多态性,程序员可以处理普遍问题而让执行环境处理特殊问题。即使在不知道对象类型的情况下,程序员也可以令各种对象表现出适合这些对象的行为。

**软件工程知识 10.6** 多态性提高了可扩展性:编写处理多态性行为的软件可以独立于接收这些命令的对象类型之外。把能够响应现有命令的新类型对象添加到原有系统,不必修改原有系统就可以实现。除实例化新对象的客户代码需要重新编译外,程序是无须重新编译的。

**软件工程知识 10.7** 抽象类为类层次结构中的各成员定义接口。抽象类中包含了要在派生类中定义的纯虚拟函数,该层次结构中的所有函数都可通过多态性使用同样的接口。

尽管不能实例化抽象类,但是可以声明抽象基类的指针和引用。实例化具体类的对象时,这些指针和引用就可用于实现派生类对象的多态性操作。

下面来看看多态性和虚拟函数的应用实例。一个屏幕管理程序需要显示许多不同类的对象,包括屏幕管理程序完成之后加到系统中的新对象类型。系统需要显示各种各样的形状(基类是 Shape)例如正方形、圆形、三角形、矩形、点、线等(每一类都是从基类 Shape 派生类)。屏幕管理程序使用基类指针或引用(指向 Shape)来管理所有要显示的类。为了能绘制所有对象(无论是在继承层次结构的哪一层),屏幕管理程序使用了一个基类指针(或引用)指向对象,这样一来只需简单地向这个对象发一条 draw 命令即可。Draw 函数在基类 Shape 里被声明为纯虚拟函数,并且在每个派生类中都被重定义。每个 Shape 对象都知道如何绘制自己。屏幕管理程序不需要担心对象是哪个类型的,或屏幕管理程序是否认识这个对象的类型——屏幕管理程序只需要简单地告诉每个对象绘制即可。

多态性特别适合实现分层的软件系统。如操作系统里,每种物理设备的操作都可能不同。尽管这样,从设备“读”(read)“写”(write)数据的命令在某种程度上可以统一。发给设备对象的“写”(write)命令需要在设备驱动程序的上下文中得以具体解释,并且要解释驱动程序如何操纵特定类型的设备。然而,write 命令本身和系统“写”(write)到其他设备的行为并没有什么区别——只是从内存中取出一些字节放到设备中。面向对象的操作系统可以将抽象基类用于所有设备驱动提供合适的接口。然后通过继承抽象基类生成执行类似操作的派生类。驱动设备提供的功能(public 接口)在基类中作为纯虚拟函数,这些纯虚拟函数的具体实现由派生类提供以便响应特定的设备驱动程序。

利用多态性编程,程序可以遍历一个容器,如一个指向类层次结构不同层的对象的指针数组。在类似的数组中,指针均指向派生类对象的基类指针。例如,TwoDimensionShape 类的一个对象数组可以包含指向派生类 Square, Circle, Triangle, Rectangle 和 Line 等等的 TwoDimensionalShape \* 指针。使用多态性时,执行绘制数组中每个对象的命令即可在屏幕上绘制出正确图形。

## 10.6 案例分析:使用多态性的工资发放系统

图 10.1 中的程序使用虚拟函数和多态性根据职工的类型执行工资计算。所用的基类是 Employee 类,它的派生类有 Boss,无论工作时间多少他都能得到固定的周薪;Commision-Worker 类对应的职工得到的是基本工资加上一定的销售额;PieceWorker 类对应的计件工根据生产的产品件数获得报酬;HourlyWorker 类对应的小时工按小时获得报酬,外加加班费。

函数 earnings 的调用当然适用于所有的职员。但是每个职员收入的计算方式都由职工类 Employee 和基类 Employee 派生而来的类共同决定。因此 earning 在基类 Employee 中被声明为纯虚拟函数,合适的计算 earning 方法由每个派生类提供。为了能计算任意一类职工的收入,程序只用了一个基类指针(或引用)指向职工对象并且调用 earnings 函数。在实际的工资发放系统中,各种职员对象可能被保存在一个数组(链表)中,数组中每个指针都是 Employee \* 类型。程序只需遍历数组中每个元素,使用 Employee \* 指针来调用每个对象的 earnings 函数。

```

1  //Fig.10.1: employ2.h
2  //Abstract base class Employee
3  #ifndef EMPLOY2_H
4  #define EMPLOY2_H
5
6  class Employee {
7  public:
8      Employee( const char *, const char * );
9      ~Employee();    //destructor reclaims memory
10     const char *getFirstName() const;
11     const char *getLastName() const;
12
13     //Pure virtual function makes Employee abstract base class
14     virtual double earnings() const = 0;    //pure virtual
15     virtual void print() const;            //virtual
16 private:
17     char *firstName;
18     char *lastName;
19 };
20
21 #endif

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——employ2.h

```

22 //Fig.10.1: employ2.cpp
23 //Member function definitions for
24 //abstract base class Employee.
25 //Note: No definitions given for pure virtual functions.
26 #include <iostream>
27
28 using std::cout;
29
30 #include <cstring>

```

```

31 #include <cassert>
32 #include "employ2.h"
33
34 //Constructor dynamically allocates space for the
35 //first and last name and uses strcpy to copy
36 //the first and last names into the object.
37 Employee::Employee( const char *first, const char *last )
38 {
39     firstName = new char[ strlen( first ) + 1 ];
40     assert( firstName != 0 );    //test that new worked
41     strcpy( firstName, first );
42
43     lastName = new char[ strlen( last ) + 1 ];
44     assert( lastName != 0 );    //test that new worked
45     strcpy( lastName, last );
46 }
47
48 //Destructor deallocates dynamically allocated memory
49 Employee::~Employee()
50 {
51     delete [] firstName;
52     delete [] lastName;
53 }
54
55 //Return a pointer to the first name
56 //Const return type prevents caller from modifying private
57 //data. Caller should copy returned string before destructor
58 //deletes dynamic storage to prevent undefined pointer.
59 const char *Employee::getFirstName() const
60 {
61     return firstName;    //caller must delete memory
62 }
63
64 //Return a pointer to the last name
65 //Const return type prevents caller from modifying private
66 //data. Caller should copy returned string before destructor
67 //deletes dynamic storage to prevent undefined pointer.
68 const char *Employee::getLastName() const
69 {
70     return lastName;    //caller must delete memory
71 }
72
73 //Print the name of the Employee
74 void Employee::print() const
75     | cout << firstName << ' ' << lastName; }

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——employ2.cpp

```

76 //Fig. 10.1: boss1.h
77 //Boss class derived from Employee
78 #ifndef BOSS1_H
79 #define BOSS1_H
80 #include "employ2.h"

```



```

81
82 class Boss : public Employee {
83 public:
84     Boss( const char *, const char *, double = 0.0 );
85     void setWeeklySalary( double );
86     virtual double earnings() const;
87     virtual void print() const;
88 private:
89     double weeklySalary;
90 };
91
92 #endif

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——boss1.h

```

93 //Fig. 10.1; boss1.cpp
94 //Member function definitions for class Boss
95 #include <iostream>
96
97 using std::cout;
98
99 #include "boss1.h"
100 101 //Constructor function for class Boss
102 Boss::Boss( const char *first, const char *last, double s )
103     : Employee( first, last ) //call base-class constructor
104 { setWeeklySalary( s ); }
105
106 //set the Boss's salary
107 void Boss::setWeeklySalary( double s )
108     { weeklySalary = s > 0 ? s : 0; }
109
110 //get the Boss's pay
111 double Boss::earnings() const { return weeklySalary; }
112
113 //Print the Boss's name
114 void Boss::print() const
115 {
116     cout << "\n          Boss: ";
117     Employee::print();
118 }

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——boss1.cpp

```

119 //Fig. 10.1; commis1.h
120 //CommissionWorker class derived from Employee
121 #ifndef COMMIS1_H
122 #define COMMIS1_H
123 #include "employ2.h"
124
125 class CommissionWorker : public Employee {
126 public:
127     CommissionWorker( const char *, const char *,

```

```

128         double = 0.0, double = 0.0,
129         int = 0 );
130     void setSalary( double );
131     void setCommission( double );
132     void setQuantity( int );
133     virtual double earnings() const;
134     virtual void print() const;
135 private:
136     double salary;           //base salary per week
137     double commission;      //amount per item sold
138     int quantity;           //total items sold for week
139 };
140
141 #endif

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——commis1.h

```

142 //Fig.10.1: commis1.cpp
143 //Member function definitions for class CommissionWorker
144 #include <iostream>
145
146 using std::cout;
147
148 #include "commis1.h"
149
150 //Constructor for class CommissionWorker
151 CommissionWorker::CommissionWorker( const char *first,
152     const char *last, double s, double c, int q )
153     : Employee( first, last ) //call base-class constructor
154 {
155     setSalary( s );
156     setCommission( c );
157     setQuantity( q );
158 }
159
160 //set CommissionWorker's weekly base salary
161 void CommissionWorker::setSalary( double s )
162     { salary = s > 0 ? s : 0; }
163
164 //set CommissionWorker's commission
165 void CommissionWorker::setCommission( double c )
166     { commission = c > 0 ? c : 0; }
167
168 //set CommissionWorker's quantity sold
169 void CommissionWorker::setQuantity( int q )
170     { quantity = q > 0 ? q : 0; }
171
172 //Determine CommissionWorker's earnings
173 double CommissionWorker::earnings() const
174     { return salary + commission * quantity; }
175

```

```

176 //Print the CommissionWorker's name
177 void CommissionWorker::print() const
178 {
179     cout << " \nCommission worker: ";
180     Employee::print();
181 }

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——commis1.cpp

```

182 //Fig.10.1: piece1.h
183 //PieceWorker class derived from Employee
184 #ifndef PIECE1_H
185 #define PIECE1_H
186 #include "employ2.h"
187
188 class PieceWorker : public Employee {
189 public:
190     PieceWorker( const char *, const char *,
191                 double = 0.0, int = 0);
192     void setWage( double );
193     void setQuantity( int );
194     virtual double earnings() const;
195     virtual void print() const;
196 private:
197     double wagePerPiece; //wage for each piece output
198     int quantity;        //output for week
199 };
200
201 #endif

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——piece1.h

```

202 //Fig.10.1: piece1.cpp
203 //Member function definitions for class PieceWorker
204 #include <iostream>
205
206 using std::cout;
207
208 #include "piece1.h"
209
210 //Constructor for class PieceWorker
211 PieceWorker::PieceWorker( const char *first, const char *last,
212                           double w, int q )
213     : Employee( first, last ) //call base-class constructor
214 {
215     setWage( w );
216     setQuantity( q );
217 }
218
219 //set the wage
220 void PieceWorker::setWage( double w )
221     { wagePerPiece = w > 0 ? w : 0; }

```

```

222
223 //set the number of items output
224 void PieceWorker::setQuantity( int q )
225     { quantity = q > 0 ? q : 0; }
226
227 //Determine the PieceWorker's earnings
228 double PieceWorker::earnings() const
229     { return quantity * wagePerPiece; }
230
231 //Print the PieceWorker's name
232 void PieceWorker::print() const
233 {
234     cout << "\n    Piece worker: ";
235     Employee::print();
236 }

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——piece1.cpp

```

237 //Fig. 10.1: hourly1.h
238 //Definition of class HourlyWorker
239 #ifndef HOURLY1_H
240 #define HOURLY1_H
241 #include "employ2.h"
242
243 class HourlyWorker : public Employee {
244 public:
245     HourlyWorker( const char *, const char *,
246                 double = 0.0, double = 0.0);
247     void setWage( double );
248     void setHours( double );
249     virtual double earnings() const;
250     virtual void print() const;
251 private:
252     double wage;    //wage per hour
253     double hours;  //hours worked for week
254 };
255
256 #endif

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——hourly1.h

```

257 //Fig. 10.1: hourly1.cpp
258 //Member function definitions for class HourlyWorker
259 #include <iostream>
260
261 using std::cout;
262
263 #include "hourly1.h"
264
265 //Constructor for class HourlyWorker
266 HourlyWorker::HourlyWorker( const char *first,
267                             const char *last,

```

```

268             double w, double h )
269 : Employee( first, last )    //call base-class constructor
270 {
271     setWage( w );
272     setHours( h );
273 }
274
275 //set the wage
276 void HourlyWorker::setWage( double w )
277     { wage = w > 0 ? w : 0; }
278
279 //set the hours worked
280 void HourlyWorker::setHours( double h )
281     { hours = h >= 0 && h < 168 ? h : 0; }
282
283 //get the HourlyWorker's pay
284 double HourlyWorker::earnings() const
285 {
286     if ( hours <= 40 ) //no overtime
287         return wage * hours;
288     else                //overtime is paid at wage * 1.5
289         return 40 * wage + ( hours - 40 ) * wage * 1.5;
290 }
291
292 //Print the HourlyWorker's name
293 void HourlyWorker::print() const
294 {
295     cout << "\n    Hourly worker: ";
296     Employee::print();
297 }

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——hourly1. cpp

```

298 //Fig. 10.1: fig10_01.cpp
299 //Driver for Employee hierarchy
300 #include <iostream>
301
302 using std::cout;
303 using std::endl;
304
305 #include <iomanip>
306
307 using std::ios;
308 using std::setiosflags;
309 using std::setprecision;
310
311 #include "employ2.h"
312 #include "boss1.h"
313 #include "commis1.h"
314 #include "piecel.h"
315 #include "hourly1.h"

```

```

316
317 void virtualViaPointer( const Employee * );
318 void virtualViaReference( const Employee & );
319
320 int main()
321 |
322     //set output formatting
323     cout << setiosflags( ios::fixed | ios::showpoint )
324         << setprecision( 2 );
325
326     Boss b( "John", "Smith", 800.00 );
327     b.print(); //static binding
328     cout << " earned $" << b.earnings(); //static binding
329     virtualViaPointer( &b ); //uses dynamic binding
330     virtualViaReference( b ); //uses dynamic binding
331
332     CommissionWorker c( "Sue", "Jones", 200.0, 3.0, 150 );
333     c.print(); //static binding
334     cout << " earned $" << c.earnings(); //static binding
335     virtualViaPointer( &c ); //uses dynamic binding
336     virtualViaReference( c ); //uses dynamic binding
337
338     PieceWorker p( "Bob", "Lewis", 2.5, 200 );
339     p.print(); //static binding
340     cout << " earned $" << p.earnings(); //static binding
341     virtualViaPointer( &p ); //uses dynamic binding
342     virtualViaReference( p ); //uses dynamic binding
343
344     HourlyWorker h( "Karen", "Price", 13.75, 40 );
345     h.print(); //static binding
346     cout << " earned $" << h.earnings(); //static binding
347     virtualViaPointer( &h ); //uses dynamic binding
348     virtualViaReference( h ); //uses dynamic binding
349     cout << endl;
350     return 0;
351 }
352
353 //Make virtual function calls off a base - class pointer
354 //using dynamic binding.
355 void virtualViaPointer( const Employee *baseClassPtr )
356 {
357     baseClassPtr -> print();
358     cout << " earned $" << baseClassPtr -> earnings();
359 }
360
361 //Make virtual function calls off a base - class reference
362 //using dynamic binding.
363 void virtualViaReference( const Employee &baseClassRef )
364 {
365     baseClassRef.print();
366     cout << " earned $" << baseClassRef.earnings();

```

367 }

输出结果:

```

Boss: John Smith earned $ 800.00
Boss: John smith earned $ 800.00
Boss: John Smith earned $ 800.00
Commission worker: Sue Jones earned $ 650.00
Commission worker: Sue Jones earned $ 650.00
Commission worker: Sue Jones earned $ 650.00
Piece worker: Bob Lewis earned $ 500.00
Piece worker: Bob Lewis earned $ 500.00
Piece worker: Bob Lewis earned $ 500.00
Hourly worker: Karen Price earned $ 500.00
Hourly worker: Karen price earned $ 500.00
Hourly worker: Karen Price earned $ 500.00

```

图 10.1 用 Employee 类的层次结构说明多态性的应用——fig10\_01.cpp

下面来看看 Employee 类(第 1~75 行)。它的 public 成员函数包括:构造函数(带有姓和名两个参数)、析构函数(用来释放动态分配的内存)、一个“get”函数返回姓、一个“get”函数返回名、纯虚拟函数 earnings 和虚拟函数 print。为什么要把 earnings 函数声明为纯虚拟函数呢?因为在 Employee 类中提供这个函数的实现没有任何意义。我们不能计算泛指的那个的职员收入。首先必须知道这个职员属于哪一类。将这个函数声明为纯虚拟函数,是说明我们要在派生类而不是基类本身提供这个函数的具体实现细节。程序员不打算在抽象基类 Employee 里调用这个纯虚拟函数;所有的派生类都会根据这些类相应的实现重定义 earnings。

基类 Boss(第 76~118 行)通过 public 方式从 Employee 类派生而来。它的 public 成员函数包括:构造函数,以姓、名和周薪为参数,将姓、名传递给 Employee 的构造函数来初始化派生类对象的基类部分的 Firstname 和 Lastname 数据成员;“set”函数,用于为 private 数据成员 weeklySalary 赋新值;虚拟函数 earnings,用于定义如何计算 Boss 的收入;虚拟函数 print 用于输出职员的类型,然后调用 Employee::print() 输出职员的姓名。

CommissionWorker 类(第 119~181 行)通过 public 方式从 Employee 派生而来。它的 public 成员函数包括:构造函数,以姓、名、基本工资、提成百分比和销售量作为参数,将姓、名传递给 Employee 的构造函数;“set”函数,用于为 private 数据成员 Salary、Commission 和 Quantity 赋值;虚拟函数 earnings,用于定义如何计算 CommissionWorker 的收入;虚拟函数 print,用于输出职员的类型,然后调用 Employee::print() 输出职员的姓名。

PieceWorker 类(第 182~236 行)通过 public 方式从 Employee 派生而来。它的 public 成员函数包括:构造函数,以姓、名、每件产品的薪水、生产产品的数量作为参数,将姓、名传递给 Employee 的构造函数;“set”函数用于为 private 数据成员 WageperPiece 和 Quantity 赋值;虚拟函数 earnings,用于定义如何计算 PieceWorker 的收入;虚拟函数 print,用于输出职员的类型,然后调用 Employee::print() 来输出职员的姓名。

HourlyWorker 类(第 237~297 行)通过 public 方式从 Employee 派生而来。它的 public 成员函数包括:构造函数,以姓、名、每小时的薪水和工作时间作为参数,将姓、名传递给 Em-

ployee 的构造函数;“set”函数,用于为 private 数据成员每小时的薪水 Wage 和 Hours 赋值;虚拟函数 earnings,用于定义如何计算 HourlyWorker 的收入;虚拟函数 print,用于输出职员类型,然后调用 Employee::print() 来输出职员的姓名。

第 298 ~ 367 行是一段驱动程序,main 中的 4 段代码非常相似,所以我们只讨论第一段处理 Boss 对象的代码。

第 326 行

```
Boss b("Jonhn","Smith",800.00)
```

实例化了 boss 类的派生类对象 b,并为构造函数提供了参数,其中包括姓、名和固定的周薪。

第 327 行

```
b.print();//static binding
```

用圆点成员选择运算符(.)显式调用 boss 版本的成员函数 print。因为在编译时可知道被调用函数的对象类型,所以属于静态绑定。使用类似的调用是便于与使用动态绑定调用正确的 print 函数进行比较。

第 328 行

```
count << "earned $" << b.earnings();//static binding
```

用圆点成员选择运算符(.)显式调用 boss 版本的成员函数 earnings。这也属于静态绑定,因为在编译时可知道被调用函数的对象类型。使用类似的调用是便于与使用动态绑定调用正确的 earnings 函数进行比较。

第 329 行

```
virtualViaPointer(&b);//uses dynamic binding
```

用派生类对象 b 的地址调用函数 virtualViaPointer(第 355 行)。函数在参数 baseClassPtr 中接收这个地址,该参数被声明为 const Employee\*。这是实现多态性的行为。

第 357 行

```
baseClassPtr->print();
```

调用 baseClassPtr 所指向对象的成员函数 print。因为 print 函数在基类中被声明为 Virtual,因此系统调用派生类对象的 print 函数——这也是多态性的做法。类似的函数调用是——如动态绑定,即通过基类指针调用虚拟函数,以便在执行时决定调用哪个函数。

第 358 行

```
count << "earned $" << baseClassPtr->earnings();
```

调用 baseClassPtr 指向对象的成员函数 earnings。因为 earnings 函数在基类中被声明为 Virtual,所以系统调用了派生类对象的 earnings 函数,这也属于动态绑定。

第 330 行

```
virtualViaReference(b);//use dunamic binding
```

调用函数 virtualViaReference(b)(第 363 行),以表明可用基类引用来调用虚拟函数实现多态性。该函数通过参数 baseClassRef 接收对象 b,该参数被声明为 const Employee&。这便是通过引用来影响多态性行为。

第 365 行

```
baseClassRef.print();
```

调用 baseClassRef 所引用的成员函数 print。因为 print 函数在基类中被声明为 Virtual,所以系统调用派生类对象的 print 函数,这也是多态性的做法。类似的函数调用属于动态绑定(即通过基类引用来调用虚拟函数,以便在执行时决定要调用的函数)。

第 366 行

```
count << "earned $" << baseClassRef.earnings();
```



调用 `baseClassPtr` 所引用对象的成员函数 `earnings`。因为 `earnings` 函数在基类中被声明为 `Virtual`，所以系统调用的是派生类对象的 `earnings` 函数，这也是动态绑定的做法。

## 10.7 新类和动态绑定

事先无法确定所有可能的类时，多态性和虚拟函数会运行良好。如有新类加入系统时，它们照样能正常运行。动态绑定允许增加新类（也叫滞后联编）。对于准备编译的虚拟函数调用，不必在编译时知道对象类型。在运行时，虚拟函数调用会被调用对象的相应成员函数匹配。

屏幕保护程序不需要重新编译就可以处理添加到系统中的新的显示对象。函数 `draw` 的调用保持不变。新的对象本身具有实际绘制的功能。这样就可以很容易地为系统增加新功能而尽量避免其他影响，同时也提高软件的可重用性。

动态绑定可以使独立软件开发商在不公开源代码的情况下发行软件。发行的软件可以只包括头文件和对象文件，不必包含源代码。软件开发者可以利用继承机制从独立软件开发商提供的类中派生出新类。派生类可以和独立软件开发商提供的类一起运行，也可以通过动态绑定在这些派生类中改写虚拟函数。

10.9 节将介绍综合的多态性案例分析。10.10 节将深入探讨如何在 C++ 中实现多态性、虚拟函数与动态绑定。

## 10.8 虚拟析构函数

在类层次结构中，用多态性动态分配对象时会出现一个问题。如果一个对象（带有非虚析构函数）被 `delete` 操作符作用于指向对象的基类指针从而显式删除该对象，基类析构函数（与该指针类型相匹配的）仍然会被这些对象调用。不管基类指针指向哪种对象类型，不管各个类的析构函数是否相同，都会出现这个问题。

解决该问题有一个简便办法——声明一个虚拟的基类析构函数。这样可以使所有的派生类析构函数变成虚拟函数，即使派生类的析构函数和基类的析构函数名称不同。这样，如果用 `delete` 操作符作用于指向派生类对象的基类指针从而显式删除类层次结构的对象，就会调用合适的类的析构函数。记住，派生类对象被删除时，派生类的基类部分也会被删除——在派生类析构函数执行之后自动执行基类的析构函数。

**良好编程习惯 10.2** 如果一个类有虚拟函数，即使该类不需要虚析构函数也应该给它提供一个虚拟的析构函数。这样可保证该类派生出来的类所包括的析构函数能被正确调用。

**常见编程错误 10.2** 构造函数不能为“虚拟的”。将构造函数声明为虚拟函数会产生语法错误。

## 10.9 案例分析：继承接口和实现

图 10.2 中的程序将重新考察前一章的 `Point`、`Circle` 和 `Cylinder` 类层次结构，不过现在将该层次结构的顶部设为一个抽象基类 `Shape`。`Shape` 类有两个纯虚拟函数 `printShapeName` 和 `print`，所以它是个抽象基类。`Shape` 还包括其他两个虚拟函数 `area` 和 `volumn`，它们都有默认

的实现方式,返回值为 0。Point 是从 Shape 类继承这两个函数的实现,在这里它们是有意义的,因为点的面积和体积都是 0。Circle 类从 Point 类那里继承了 volume 函数,但 Circle 本身提供了函数 Volume 的实现。Cylinder 为函数 area 和 volume 提供了自己的实现方法。

```

1 //Fig.10.2: shape.h
2 //Definition of abstract base class Shape
3 #ifndef SHAPE_H
4 #define SHAPE_H
5
6 class Shape {
7 public:
8     virtual double area() const { return 0.0; }
9     virtual double volume() const { return 0.0; }
10
11     //pure virtual functions overridden in derived classes
12     virtual void printShapeName() const = 0;
13     virtual void print() const = 0;
14 };
15
16 #endif

```

图 10.2 用 Shape 类层次结构说明接口继承方式——shape.h

```

17 //Fig.10.2: point1.h
18 //Definition of class Point
19 #ifndef POINT1_H
20 #define POINT1_H
21
22 #include <iostream>
23
24 using std::cout;
25
26 #include "shape.h"
27
28 class Point : public Shape {
29 public:
30     Point( int = 0, int = 0 ); //default constructor
31     void setPoint( int, int );
32     int getX() const { return x; }
33     int getY() const { return y; }
34     virtual void printShapeName() const { cout << "Point: "; }
35     virtual void print() const;
36 private:
37     int x, y; //x and y coordinates of Point
38 };
39
40 #endif

```

图 10.2 用 Shape 类层次结构说明接口继承方式——point1.h

```

41 //Fig.10.2: point1.cpp
42 //Member function definitions for class Point

```

```

43 #include "point1.h"
44
45 Point::Point( int a, int b ) { setPoint( a, b ); }
46
47 void Point::setPoint( int a, int b )
48 {
49     x = a;
50     y = b;
51 }
52
53 void Point::print() const
54     | cout << '[' << x << ", " << y << ']' ; }

```

图 10.2 用 Shape 类层次结构说明接口继承方式——point1. cpp

```

55 //Fig.10.2: circle1.h
56 //Definition of class Circle
57 #ifndef CIRCLE1_H
58 #define CIRCLE1_H
59 #include "point1.h"
60
61 class Circle : public Point {
62 public:
63     //default constructor
64     Circle( double r = 0.0, int x = 0, int y = 0 );
65
66     void setRadius( double );
67     double getRadius() const;
68     virtual double area() const;
69     virtual void printShapeName() const { cout << "Circle: "; }
70     virtual void print() const;
71 private:
72     double radius;    //radius of Circle
73 };
74
75 #endif

```

图 10.2 用 Shape 类层次结构说明接口继承方式——circle1. h

```

76 //Fig.10.2: circle1.cpp
77 //Member function definitions for class Circle
78 #include <iostream>
79
80 using std::cout;
81
82 #include "circle1.h"
83
84 Circle::Circle( double r, int a, int b )
85     : Point( a, b ) //call base-class constructor
86 { setRadius( r ); }
87
88 void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }

```

```

89
90 double Circle::getRadius() const { return radius; }
91
92 double Circle::area() const
93     { return 3.14159 * radius * radius; }
94
95 void Circle::print() const
96 {
97     Point::print();
98     cout << "Radius = " << radius;
99 }

```

图 10.2 用 Shape 类层次结构说明接口继承方式——circle1.cpp

```

100 //Fig. 10.2: circle1.h
101 //Definition of class Circle
102 #ifndef CIRCLE1_H
103 #define CIRCLE1_H
104 #include "circle1.h"
105
106 class Cylinder : public Circle{
107 public:
108     //default constructor
109     Circle( double h = 0.0, double r = 0.0,
110           int x = 0, int y = 0 );
111
112     void setRadius( double );
113     double getRadius() const;
114     virtual double area() const;
115     virtual double volume() const;
116     virtual void printShapeName() const { cout << "Circle: "; }
117     virtual void print() const;
118 private:
119     double radius;    //radius of Circle
120 };
121
122 #endif

```

图 10.2 用 Shape 类层次结构说明接口继承方式——cylinder1.h

```

123 //Fig. 10.2: cylindr1.cpp
124 //Member and friend function definitions for class Cylinder
125 #include <iostream>
126
127 using std::cout;
128
129 #include "cylindr1.h"
130
131 Cylinder::Cylinder( double h, double r, int x, int y )
132     : Circle( r, x, y ) //call base-class constructor
133 { setHeight( h ); }
134

```

```
135 void Cylinder::setHeight( double h )
136     { height = h > 0 ? h : 0; }
137
138 double Cylinder::getHeight() { return height; }
139
140 double Cylinder::area() const
141 {
142     //surface area of Cylinder
143     return 2 * Circle::area() +
144         2 * 3.14159 * getRadius() * height;
145 }
146
147 double Cylinder::volume() const
148     { return Circle::area() * height; }
149
150 void Cylinder::print() const
151 {
152     Circle::print();
153     cout << "; Height = " << height;
154 }
```

图 10.2 用 Shape 类层次结构说明接口继承方式——cylinder1.cpp

```
155 //Fig. 10.2; fig10_02.cpp
156 //Driver for shape, point, circle, cylinder hierarchy
157 #include <iostream>
158
159 using std::cout;
160 using std::endl;
161
162 #include <iomanip>
163
164 using std::ios;
165 using std::setiosflags;
166 using std::setprecision;
167
168 #include "shape.h"
169 #include "point1.h"
170 #include "circle1.h"
171 #include "cylindr1.h"
172
173 void virtualViaPointer( const Shape * );
174 void virtualViaReference( const Shape & );
175
176 int main()
177 {
178     cout << setiosflags( ios::fixed | ios::showpoint )
179         << setprecision( 2 );
180
181     Point point( 7, 11 );           //create a Point
182     Circle circle( 3.5, 22, 8 );    //create a Circle
```

```

183   Cylinder cylinder( 10, 3.3, 10, 10 ); //create a Cylinder
184
185   point.printShapeName();    //static binding
186   point.print();             //static binding
187   cout << '\n';
188
189   circle.printShapeName();   //static binding
190   circle.print();            //static binding
191   cout << '\n';
192
193   cylinder.printShapeName(); //static binding
194   cylinder.print();          //static binding
195   cout << "\n\n";
196
197   Shape * arrayOfShapes[ 3 ]; //array of base-class pointers
198
199   //aim arrayOfShapes[0] at derived-class Point object
200   arrayOfShapes[ 0 ] = &point;
201
202   //aim arrayOfShapes[1] at derived-class Circle object
203   arrayOfShapes[ 1 ] = &circle;
204
205   //aim arrayOfShapes[2] at derived-class Cylinder object
206   arrayOfShapes[ 2 ] = &cylinder;
207
208   //Loop through arrayOfShapes and call virtualViaPointer
209   //to print the shape name, attributes, area, and volume
210   //of each object using dynamic binding.
211   cout << "Virtual function calls made off "
212         << "base-class pointers\n";
213
214   for ( int i = 0; i < 3; i++ )
215       virtualViaPointer( arrayOfShapes[ i ] );
216
217   //Loop through arrayOfShapes and call virtualViaReference
218   //to print the shape name, attributes, area, and volume
219   //of each object using dynamic binding.
220   cout << "Virtual function calls made off "
221         << "base-class references\n";
222
223   for ( int j = 0; j < 3; j++ )
224       virtualViaReference( *arrayOfShapes[ j ] );
225
226   return 0;
227 }
228
229 //Make virtual function calls off a base-class pointer
230 //using dynamic binding.
231 void virtualViaPointer( const Shape *baseClassPtr )
232 {
233     baseClassPtr->printShapeName();

```

```

234     baseClassPtr->print();
235     cout << " \nArea = " << baseClassPtr->area()
236         << " \nVolume = " << baseClassPtr->volume() << " \n\n";
237 }
238
239 //Make virtual function calls off a base-class reference
240 //using dynamic binding.
241 void virtualViaReference( const Shape &baseClassRef )
242 |
243     baseClassRef.printShapeName();
244     baseClassRef.print();
245     cout << " \nArea = " << baseClassRef.area()
246         << " \nVolume = " << baseClassRef.volume() << " \n\n";
247 }

```

输出结果:

```

Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

```

图 10.2 用 Shape 类层次结构说明接口继承方式——fig10\_02. cpp

**软件工程知识 10.8** 一个类可以从基类继承接口和/或实现。为实现继承而设计的层次结构倾向于在高层上实现具体功能,即每个新派生类都继承基类所定义的一个或多个成员函数,并使用基类的定义。为实现接口继承而设计的层次结构则倾向于在低层实现具体功能,即基类定义一个或几个函数,在层次结构中每个对象都一样调用它们(即有同样的签名),但各个派生类自己提供这些函数的实现方案。

基类 Shape (第 1 ~ 16 行) 包括 4 个 public 虚拟函数,不包括任何数据。函数 printShapeName 和 print 是纯虚拟函数,因此它们需要在派生类中重新定义。函数 area 和 vol-

umn 被定义返回 0.0。当派生类需要用不同方式计算面积( area)和体积( volumn)时,需要在派生类中重新定义它们。注意,Shape 是一个抽象类,它包括一些“不纯”的虚拟函数( area 和 volumn)。抽象类也可以包括非虚拟函数和由派生类继承来的数据。

Point 类(第 17 ~ 54 行)通过 public 继承方式由 Shape 类派生而来。Point 的面积和体积均为 0.0,因此基类成员函数 area 和 volumn 在这里并未被改写(只是简单继承了在 Shape 类中的定义)。函数 printShapeName 和 print 是基类中被定义为纯虚的虚拟函数的具体实现。如果 Point 类里没有改写它们,那 Point 类也是个抽象类,我们也不能实例化 Point 对象。Point 类还包括其他一些成员函数,分别是:用来给一个 Point 的 x 坐标和 y 坐标赋新值的“set”函数;用来返回 Point 的 x 坐标和 y 坐标的“get”函数。

Circle 类(第 55 ~ 99 行)通过 public 方式从 Point 类派生而来。Circle 的体积是 0.0,所以基类成员函数 volumn 此时没有被改写而是从 Point 继承而来,而 Point 的 volumn 函数从 Shape 类继承而来。Circle 的而积是非零的。因此这里需要改写 area 函数。函数 printShapeName 和 print 是基类 Shape 中被定义成纯虚的虚拟函数的具体实现。我们在 point 类里没有改写它们,所以将从 Point 版本的函数继承。Circle 类还包括其他一些成员函数,如用于为 Circle 的 radius 赋新值的“set”函数;用于返回 radius 的“get”函数。

Cylinder 类(第 100 ~ 154 行)通过 public 方式从 Circle 类派生而来。Cylinder 的而积和体积与 Circle 都不相同,所以这里要改写 area 函数和 volumn 函数。函数 printShapeName 和 print 是基类 Shape 中被定义为纯虚的虚拟函数的具体实现。我们在 Point 类里没有改写它们,所以将从 Circle 版本的函数继承。类 Cylinder 还包括其他一些成员函数,分别是:用于为 Height 赋新值的有“set”函数;用于返回 Height 的“get”函数。

驱动程序(第 155 ~ 247 行)首先实例化 Point 对象 point、Circle 对象 circle 和 Cylinder 对象 cylinder,然后调用函数 printShapeName 和 print 输出每个对象的名字并验证每个对象是否实例化成功。第 185 ~ 194 行对 printShapeName 和 print 的调用都是静态绑定,即在编译时就知道调用 printShapeName 和 print 的每种对象的类型。

接着声明一个指针数组 arrayOfShapes,它的每个元素都是 Shape \* 类型的。这个基类指针数组用于指向每个派生类对象。第 200 行对象 point 的地址赋给了 arrayOfShape[0]、第 203 行将 circle 的地址赋给 arrayOfShape[1],第 206 行 cylinder 的地址赋给 arrayOfShape[2]。

第 214 行用 for 结构遍历数组 arrayOfShape,并在第 215 行

```
VirtualViaPointer(arrayOfShapes[i]);
```

对每个数组元素调用函数 virtualViaPoint。函数 virtualViaPointer 用 baseClassPtr(类型是 const Shape \*) 参数来接收 arrayOfShape 数组中存放的地址。每次执行 virtualViaPointer,都会调用以下 4 个虚拟函数

```
baseClassPtr->printShapeName()
baseClassPtr->print()
baseClassPtr->area()
baseClassPtr->volumn()
```

这些调用方法在执行时 baseClassPtr 所指向的对象都会调用一个虚拟函数(编译时无须知道其具体类型)。输出的结果将显示每个类调用的相应函数。首先,输出字符串“Point:”和相应的对象 Point 的坐标,其面积和体积都是 0.0。然后输出字符串“Circle:”、对象 Circle 圆点



的坐标和半径;计算 Circle 的面积并返回体积为 0.0。最后,输出字符串“Cylinder:”、对象 Cylinder 的底部圆点坐标、计算其面积和体积。所有调用 printShapeName、print、area 及 volumn 的虚拟函数都是在运行时用动态关联解决的。

最后,第 223 行用一个 for 结构遍历 arrayOfShapes 数组,并在第 224 行

```
virtualViaReference(*arrayofShapes[j]);
```

对每个数组元素调用函数 virtualViaReference。函数 virtualViaReference 通过 baseClassRef (const Shape&) 参数来接收对数组元素中存放的地址的引用。每次执行对 virtualViaReference 的调用,都会调用以下 4 个虚拟函数

```
baseClassRef.printShapeName()
baseClassRef.print()
baseClassRef.area()
baseClassRef.volumn()
```

前面这些调用方法都会调用 baseClassRef 所引用对象的相应函数。用基类引用的方式输出和使用基类指针的方式产生的结果是一样的。

## 10.10 多态性、虚拟函数和动态绑定的本质

C++ 的多态性编程比较容易实现。虽然其方式与 C 语言的非面向对象语言编程一样,但这样做会涉及到复杂而危险的指针操作。本节将讨论 C++ 内部如何实现多态性、虚拟函数和动态绑定,以便深刻了解这些功能是如何实现的。更重要的是,这有助于让你客观评价多态性的系统开销(内存占用和处理器时间除外),以便你决定何时该用多态性。第 20 章将介绍 STL 组件的实现无需使用多态性和虚拟函数,其目的是缩短执行时间,以满足 STL 对最佳性能的特别要求。

首先,我们要解释一下 C++ 编译器如何在编译时建立数据结构,以支持运行时的多态性要求。

然后我们要看一下执行中的程序如何利用这些数据结构来执行虚拟函数和与多态性相关的动态绑定。

在编译带有一个或多个虚拟函数的类时,C++ 会为该建立一个虚拟函数表(vtable)。每次执行该类的虚拟函数时,执行中的程序都会用 vtable 来选择恰当的实现方法。图 10.3 就是 Shape 类、Point 类、Circle 和 Cylinder 的虚拟函数表。

Shape 类的 vtable 中,第一个函数指针指向该类 area 函数的实现方法,即返回面积 0.0。第二个函数指针指向 volumn 函数,返回体积也是 0.0。printShapeName 和 print 函数都是纯虚拟函数(它们没有实现方法,所以函数指针均设为 0)。任何类在其 vtable 中都有一个或多个函数指针是 0,它就是抽象类。类的 vtable 没有任何为 0 的指针,该类(如 Point、Circle 和 Cylinder)就可称为具体类。

Point 类继承了 Shape 类的 area 和 volumn 函数,因此编译器只在 Point 类的 vtable 中将这两个指针设为 Shape 类的 area 和 volumn 指针的副本。Point 类改写了 printShapeName 为打印“point:”,因此该函数指针指向 Point 类的 printShapeName 函数。Point 类改写了 print,因此相应的函数指针指向 Point 类的 print 函数打印[x,y]。

Circle 类的 vtable 中, area 函数指针指向 Circle 的 area 函数用于返回  $\pi r^2$ 。Volumn 函数指针是 Point 类的副本(该指针在前面从 Shape 类复制到 Point)。PrintShapeName 函数指针指向 Circle 版的函数打印出“Circle;”。print 函数指针指向 Circle 的 print 函数,用于打印  $[x, y].r$ 。

Cylinder 类的 vtable 中, Circle area 函数指针指向 Circle area 函数,该函数将返回  $\pi r^2$ 。Volumn 函数指针(即以前从 shape 类复制到 Point 类的那个指针),将从 Point 类开始复制。PrintShapeName 函数指针指向打印“Cylinder;”的一个函数。print 函数指针指向一个打印  $[x, y].r$  的函数。

多态性是通过涉及到 3 层指针的、复杂的数据结构来实现的。我们已经讨论了一层(在 vtable 函数指针)。调用虚拟函数时,它们将指向实际执行的函数。

现在我们要讨论第二层指针。只要带有虚拟函数的类被实例化成对象了,编译器都在该对象前面加上该类的 vtable 指针(注意:这个指针通常放在对象前面,但这不作特别需求)。

第 3 层指针是接收虚拟函数调用的对象句柄(这个句柄也可以是一个引用)。

现在来看看一个典型的虚拟函数调用。以函数 virtualViaPointer 中的下列调用

```
baseClassPtr->printShapeName()
```

为例,假设 baseClassPtr 包括了在 arrayOfShape[1] 中的地址(即对象 Circle 的地址)。编译器编译这条语句时,它要确定这个调用实际上由基类指针来执行,而且 printShapeName 是一个虚拟函数。

接下来,编译器确定 printShapeName 是每个 vtable 表中的第 3 个项目。为找到该项目,编译器要跳过前两个项目。为此,编译器将 8 个字节的偏移量或位移量(在目前流行的 32 位机器中,每个指针为 4 个字节)编译成机器语言的目标码,用于执行虚拟函数调用。

然后,编译器产生代码(注意:下列编号对应图 10.3 中圆圈内的数字)来完成:

(1) 从 arrayOfShape 中选择第 i 个项目(本例是对象 circle 的地址),并将其传给 virtualViaPointer,以使 baseClassPtr 指向 Circle。

(2) 复引用指针以取得 Circle 对象(以指向 circle vtable 的指针开始)。

(3) 复引用 Circle 的 vtable 指针以获得 circle vtable。

(4) 跳过 8 个字节的位移取得 printShapeName 函数的指针。

(5) 复引用 printShapeName 函数指针构成实际要执行的函数名,并用函数调用操作符()执行合适的 printShapeName 函数和打印字符串“Circle”。

图 10.3 中的数据结构看起来较复杂,但大部分都由编译器负责,程序员不必操心,C++ 可以直接实现多态性编程。

每次调用虚拟函数时,指针复引用操作和内存访问都需要增加运行时间。加入对象的 vtable 和 vtable 指针也要占用内存。

可喜的是,现在已有了关于虚拟函数操作的详细信息,可以确定其是否合适具体的应用。

**性能提示 10.1** 通过虚拟函数和动态绑定实现的多态性非常高效。程序员使用这种功能时,不会对系统性能有较大影响。

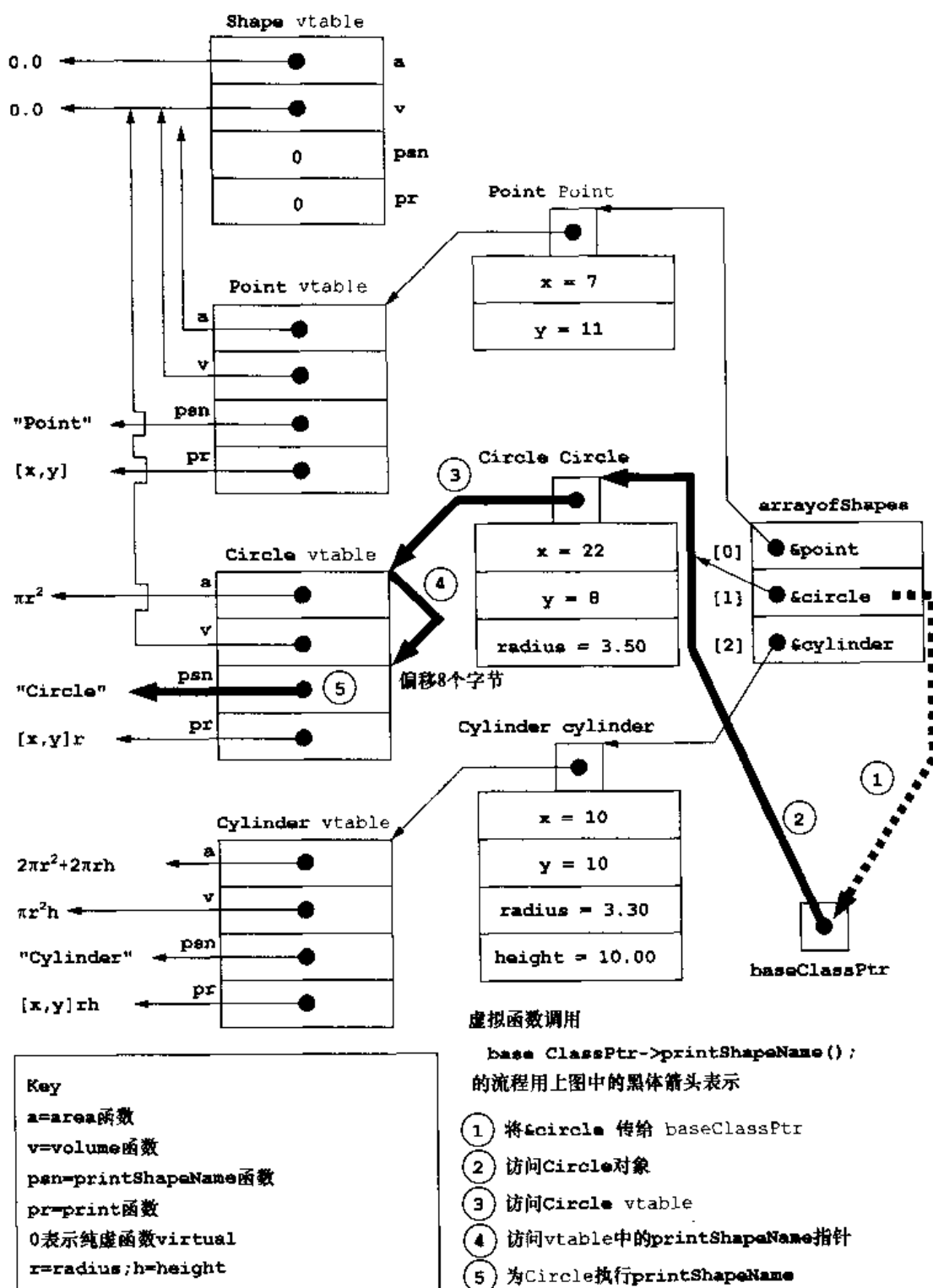


图 10.3 虚拟函数调用的程序控制流程

**性能提示 10.2** 利用虚拟函数和动态绑定,多态性编程与 switch 逻辑编程形成了鲜明对比。通常情况下,C++ 优化编译器生成的代码运行效率至少和基于 switch 的逻辑代码一样。对大多数应用来说,多态性的开销是可以接受的。但在某些情况下(如性能要求很高的实时应用),多态性的开销可能较高。

## 10.11 小结

- 使用虚拟函数和多态性设计和实现的系统可能有更高的可扩展性。在程序开发过程中,程序员可以编写程序用于处理可能不存在的对象类型。
- 使用虚拟函数和多态性编程之后可以不再使用 switch 逻辑。程序员可以使用虚拟函数自动执行等价的逻辑,这样就避免了与 switch 逻辑相关的典型错误。让客户代码来确定对象类型和表达则是不受欢迎的类设计。
- 必要时,派生类可以提供基类虚拟函数的具体实现,否则就使用基类的实现。
- 如果通过用名字和圆点成员选择操作符(.)引用特定对象来调用虚拟函数,引用是在编译时确定的(称为静态绑定),被调用的虚拟函数是特定对象的类定义的或继承该类的函数。
- 在某些情况下,定义不会用于实例化为对象的类大有用处。这样的类被称为抽象类。因为它们只能用作基类,所以也称为抽象基类。抽象基类在程序中不能被实例化为对象。
- 可以被实例化为对象的类成为具体类。
- 将类的一个或一个以上的虚拟函数声明为纯虚,该类就会成为抽象类。纯虚拟函数是在声明时被初始化为 0 的函数。
- 如果一个类是由一个带有纯虚拟函数的类派生出来的,并且该派生类没有提供对该纯虚拟函数的具体定义,那么该纯虚拟函数在该派生类中还是纯虚的。派生类也是一个抽象类。
- C++ 支持多态性,即通过继承而相关的不同类的对象能够通过对同一个成员函数的调用做出不同的反应。
- 多态性是通过虚拟函数来实现的。
- 当通过基类指针或引用来请求调用虚拟函数。C++ 会在与对象相关的派生类中选择改写正确的函数。
- 通过使用虚拟函数和多态性,对一个成员函数的调用会根据接收到该调用的对象的类型不同会做出不同的反应。
- 尽管不能实例化抽象类,但我们可以声明指向抽象类的指针。当派生的具体类被实例化为对象后,这些指针就可以用来实现这些派生类对象的多态性操作。
- 通过动态绑定(也叫滞后绑定)可以给系统增加新类。虚拟函数调用在编译时不需要知道对象的类型,在运行时会寻找匹配的对象成员函数。
- 动态绑定可以是独立软件开发商在不透露其源代码的情况下发行软件。发行的软件只包括头文件和对象文件,而不需要公布其源代码。和独立软件开发商提供的类一起运行的软件也能够和派生类一起运行,也可以(通过动态绑定)使用这些类提供的改写过的虚拟函数。
- 动态绑定要求在运行时把虚成员函数的调用定位到对应类的该虚拟函数的版本。被称为 vtable 的虚拟函数表示一个包含有函数指针的数组。每个带有虚拟函数的类

都会有一张这样的 vtable。针对该类里的每个虚拟函数, vtable 都有一个函数指针, 指向类对象相对应的该虚拟函数的版本。特定类使用虚拟函数可能是该类改写过的函数, 也可能是从层次结构中较高层的基类直接或间接继承而来的函数。

- 基类提供一个虚成员函数, 派生类可以改写这个虚拟函数, 但派生类不一定要这样做。派生类可以用基类版本的虚成员函数, 在该类的 vtable 可以指明。
- 带有虚拟函数的类的每个对象都包含一个指向该类 vtable 的指针。可以获得 vtable 中相应的函数指针, 并在运行时重引用来完成调用。对 vtable 的检索和指针重引用占用很少的运行时间, 一般少于最优的客户代码。
- 如果一个类包含虚拟函数, 就要把基类的析构函数声明为虚析构函数。这样会使所有派生类的析构函数自动成为虚析构函数, 即使它们的名称与基类析构函数不同。如果使用 delete 操作符作用于指向派生类对象的基类指针, 以显式删除层次结构中的对象, 系统就会调用相应类的析构函数。
- 任何一个类只要其 vtable 中有一个或一个以上的 0 指针, 它就是抽象类。没有 0 指针的(如 Point, Circle 和 Cylinder)则是具体类。

## 本章术语

abstract base class 抽象基类

abstract class 抽象类

base-class virtual function 基类虚拟函数

class hierarchy 类层次结构

concrete class 具体类

convert derived-class pointer to base-class pointer

将派生类指针转换为基类指针

derived class 派生类

derived-class constructor 派生类构造函数

direct base class 直接基类

displacement into vtable vtable 位移

dynamic binding 动态绑定

early binding 提前绑定

eliminating switch statements 消除 switch 语句

explicit pointer conversion 显式指针转换

extensibility 可扩展性

implementation inheritance 实现继承

independent software vendor (ISV)

独立软件供应商 (ISV)

indirect base class 间接基类

inheritance 继承

interface inheritance 接口继承

late binding 滞后绑定

offset into vtable vtable 偏移量

override a pure virtual function 改写纯虚拟函数

override a virtual function 改写虚拟函数

pointer to a base class 基类指针

pointer to a derived class 派生类指针

pointer to an abstract class 抽象类指针

polymorphism 多态性

programming "in the general" 泛型编程

programming "in the specific" 特定编程

pure virtual function (=0) 纯虚拟函数 (=0)

reference to a base class 基类引用

reference to a derived class 派生类引用

reference to an abstract class 抽象类引用

software reusability 软件可重用性

static binding 静态绑定

switch logic switch 逻辑

virtual destructor 虚析构函数

virtual function table 虚拟函数表

virtual function 虚拟函数

vtable pointer vtable 指针

## 常见编程错误

10.1 试图实例化抽象类对象(即包括一个或多个纯虚拟函数的类)会导致语法错误。

10.2 构造函数不能为“虚拟”(virtual)。将构造函数声明为虚拟函数会产生语法错误。

### 良好编程习惯

- 10.1 尽管某个函数在类层次结构中的高层中被声明为虚拟函数可使其在低层隐式成为虚拟函数,但在每层都显式声明这些虚拟函数可增强程序的可读性。
- 10.2 如果一个类有虚拟函数,即使它不需要虚拟的析构函数,也应该为它提供一个虚拟的析构函数。这样可保证该类的派生类所包括的析构函数能被正确调用。

### 性能提示

- 10.1 通过虚拟函数和动态绑定实现的多态性非常高效。程序员使用这种功能时,不会对系统性能有较大影响。
- 10.2 虚拟函数和动态绑定使得多态性编程与 switch 逻辑编程形成了鲜明对比。通常情况下C++ 优化编译器生成的代码运行的效率起码能与基于 switch 的逻辑代码一样。对大多数应用来说,多态性的开销是可以接受的。但在某些情况下(如性能要求很高的实时应用)多态性的开销可能太高。

### 软件工程知识

- 10.1 使用 virtual 函数和多态性后,一个很有趣的结果是程序看上去很简单。程序中很少有分支逻辑,而是一些简单的顺序的代码。这大大简化了测试、调试、程序维护,避免了错误。
- 10.2 函数一旦被声明为虚拟函数,即使一个类改写它时没有将其声明为虚拟函数,它从该点之后的继承层次结构中仍然是虚拟函数。
- 10.3 派生类不定义虚拟函数时,可简单继承其直接基类的虚拟函数的定义。
- 10.4 如果类从一个带有纯虚拟函数的类派生而来,且该派生类中没有提供该纯虚拟函数的定义,那么这个纯虚拟函数在该派生类中仍然是纯虚的,这个派生类仍然是抽象类。
- 10.5 利用虚拟函数和多态性,程序员可以处理普遍问题而让执行环境处理特殊问题。即使在不知道对象类型的情况下,程序员也可以令各种对象表现出适合这些对象的行为。
- 10.6 多态性提高了可扩展性:处理多态性行为的软件可以用与接收消息的对象类型无关的方式编写。这样一来,不必修改基本系统就可以把能够响应现有消息的新类型的对象添加到系统中。除实例化新对象的客户代码需要重新编译外,程序是无须重新编译的。
- 10.7 抽象类为类层次结构中的各个成员定义接口。抽象类中包含了要在派生类中定义的纯虚拟函数,该层次结构中的所有函数都可以通过多态性使用同样的接口。
- 10.8 一个类可以从基类继承接口和(或)实现。为实现继承而设计的层次结构趋向于在高层上实现具体功能,即每个新派生类都继承基类所定义的一个或多个成员函数,并使用基类的定义。为实现接口继承而设计的层次结构则趋向于在低层实现具体功能,即基类定义一个或几个函数,在层次结构中每个对象都一样调用它们(即有同样的签名),但各个派生类自己提供提供这些函数的实现方案。

### 自测题

- 10.1 填空题:

- a) 使用继承和多态性有助于逻辑\_\_\_\_\_。
- b) 在类定义时,将\_\_\_\_\_置于虚拟函数原型的末尾来声明虚拟函数。
- c) 如果一个类包含有一个或一个以上的纯虚拟函数,那么该类被称为\_\_\_\_\_。
- d) 在编译时确定函数调用被称为\_\_\_\_\_绑定。
- e) 在运行时确定函数调用被称为\_\_\_\_\_绑定。

### 自测题答案

10.1 a) switch b) =0 c) 抽象类 d) 静态或提前 e) 动态或滞后

### 练习题

- 10.2 什么是虚拟函数? 请举出一个适合使用虚拟函数的例子。
- 10.3 构造函数不能是虚拟函数,如何才能使构造函数具有虚拟函数的功能?
- 10.4 多态性是如何让程序“普遍化”而非“特殊化”的? 试说明程序“普遍化”的主要好处。
- 10.5 试说明 switch 逻辑编程存在问题,并解释多态性可以更有效替代 switch 逻辑的原因。
- 10.6 说出静态绑定与动态绑定的区别,并解释动态绑定虚拟函数和 vtable 的用法。
- 10.7 区分继承接口和继承实现。继承接口的继承层次结构设计与继承实现的层次结构设计有何不同?
- 10.8 区分虚拟函数和纯虚拟函数。
- 10.9 (判断正误)抽象类中所有的虚拟函数必须被声明为纯虚拟函数。
- 10.10 为本章中 shape 层次结构提出一层或多层的抽象基类(第一层是 Shape,第二层包括 TwoDimensionalShape 和 ThreeDimensionalShape)。
- 10.11 多态性是如何提高可扩展性的?
- 10.12 开发一个详细描述图形输出的飞行器模拟程序。解释为什么多态性特别适合用于解决这类问题?
- 10.13 开发一个基本图形包。使用第9章的 Shape 类继承层次结构,仅限于二维图形如正方形、矩形、三角形、圆。可以与用户交互,让用户指定要绘制的形状的位置、大小、形状和填充字符。用户可以指定同个形状的多个项目。在生成每个形状时,都将一个指向每个新 Shape 对象的 Shape \* 的指针放入数组。每个类都有自己的 draw 成员函数。写一个多态性的屏幕管理程序遍历这个数组(可以用迭代),给数组中每个对象发送一个 draw 命令形成屏幕图形。每当用户指定新的图形时重新绘制屏幕图形。
- 10.14 修改图 10.1 的工资发放系统,给 Employee 类增加 public 成员 birthDate(一个 date 对象)和 departmentcode(一个 int 型)。假定该系统每月处理一次。程序要计算每个 Employee(多态性的)的工资,并且如果 Employee 生日所在的月份,就给该员工增加 100 元奖金。
- 10.15 练习题 9.14 中,开发了一个 Shape 类层次结构,并且定义了其中的类。修改这个层次结构以使 Shape 类是一个包含了接口的抽象基类。从 Shape 类派生抽象类 TwoDimensionalShape 和 ThreeDimensionalShape。而且包括虚拟函数 area 和 volume 以计算类层次结构中每个具体类的对象面积和体积。写一个驱动程序测试 Shape 类的层次结构。

# 第 11 章 C++ 输入/输出流

## 学习目标

- 理解如何使用C++ 面向对象的输入/输出流
- 能够格式化输入/输出
- 理解 I/O 流类的层次结构
- 理解如何输入/输出用户自定义类型的对象
- 能够创建用户自定义的流操纵元
- 能够判断输入/输出操作是否成功
- 能够将输出流连接到输出流

## 11.1 简介

C++ 标准类库提供了一系列可扩展的输入/输出(I/O)功能。本章将详细讨论一些最常见的 I/O 操作功能,并简要概述其余的输入/输出功能。本章部分内容在前面已有介绍,但这里将全面介绍输入/输出功能。

这里描述的输入/输出功能都是面向对象的。了解C++ 输入/输出特性的实现细节是非常有趣的。C++ 实现的输入/输出功能还利用了C++ 其他一些功能,如引用、函数重载和操作符重载等。

稍后你会注意到,C++ 使用类型安全(type safe)的 I/O 操作。每个 I/O 操作符都是自动以数据类型敏感的方式执行的。如果某个 I/O 函数被定义为操纵特定的数据类型,该函数就会在必要时被自动调用以处理该类数据类型。如果实际的数据类型和函数不匹配,就会出现编译错误。因此,非法的数据不能通过系统检测。C 语言则不然这是个漏洞,会产生微妙而又奇怪的错误。

用户可以指定自定义的 I/O 类型,也可以指定标准类型的 I/O。这种扩展能力是C++ 最有价值的功能之一。

**良好编程习惯 11.1** 尽管 C 类型的 I/O 实际可用于C++ ,但在C++ 编程中尽量只用C++ 类型的 I/O。

**软件工程知识 11.1** C++ 类型的 I/O 具有类型安全性。

**软件工程知识 11.2** C++ 允许采用同样的方式处理预先定义的类型和用户自定义的类型。这样可提高软件的开发速度,尤其是提高软件的可重用性。

## 11.2 流

C++ 中的 I/O 是以字节流的形式实现的。流就是一个字节的序列。输入操作时,字节



从设备(如键盘、磁盘、网络连接)输入到内存。输出操作时,字节从内存输出到外部设备(如显示器、打印机、磁盘和网络连接)。

应用程序中的字节是有含义的。字节可以表示 ASCII 字符,内部格式的原始数据、图形图像、数字音频、数字视频或其他应用程序所需的信息。

输入/输出系统机制的任务是稳定可靠地在设备和内存之间传送字节。这一过程常常包括一些机械运动如磁盘或磁带的转动、键盘的击键。处理器处理数据的时间相比这一过程所花费的时间要多得多。因此,要想获得最佳性能,就须要对输入/输出操作进行周密的规划和调节。

C++ 提供了“低级”和“高级”的输入/输出功能。低级的 I/O 功能(如无格式的 I/O)通常指大量的字节在设备到内存或内存到设备的简单传输。这一传输过程以单个字节为单位。这种低级的输入/输出功能的确提供了高速而大量的传输。但是,使用时并不方便。

人们更喜欢 I/O 高级视图(格式化的 I/O),它将字节组合成有意义的单元,如整数、浮点数、字符、字符串和自定义类型。这些面向类型的功能可适用于绝大部分输入/输出(大容量文件处理)。

**性能提示 11.1** 对于大容量文件的处理,使用无格式的 I/O 可获得最佳性能。

### 11.2.1 iostream 类库的头文件

C++ 的 iostream 类库提供了数以百计的 I/O 功能。许多头文件都包含类库接口。

大多数 C++ 程序都包括 `<iostream>` 头文件。这个头文件声明了所有 I/O 操作所需要的基本服务。`<iostream>` 头文件定义了 `cin`, `cout`, `cerr` 和 `clog` 对象,分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。该头文件提供了格式化和无格式的 I/O 服务。

`<iomanip>` 头文件声明了对于执行格式化 I/O(即所谓的含参数流操纵元)非常有用的服务。

`<fstream>` 头文件声明了对于用户控制文件处理操作非常重要的服务。第 14 章的文件处理程序将使用该头文件。

C++ 的每个版本通常都包括其他相关 I/O 的库,这些库提供了特定系统的某些功能如控制特殊用途的音频或视频输入/输出设备。

### 11.2.2 输入/输出流类和对象

iostream 类库包含了许多用于处理大量 I/O 操作的类。istream 类支持流输入操作。ostream 类支持流输出操作。iostream 类同时支持流输入和流输出操作。

istream 类和 ostream 类都是从 ios 基类简单继承而来。iostream 类是 istream 类和 ostream 类多重派生而来。这一继承关系在图 11.1 中表示出来。

操作符重载对于执行输入/输出提供了一个方便的方法。左移位操作符得以重载以表示输入,被称为流插入操作符;右移位操作符得以重载以表示流输出,被称为流读取操作符。这些操作符可以与标准流对象 `cin`, `cout`, `cerr` 和 `clog` 及用户自定义流对象一起使用。

对象 `cin` 是预先定义的,它是 istream 类的一个实例,和标准输入设备连在一起,通常是键盘。下面语句中的流读取操作符把整型变量 `grade`(假设 `grade` 已声明为整型变量)从 `cin` 输出到内存。

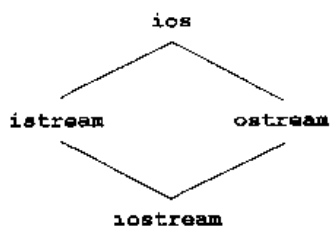


图 11.1 输入/输出流的部分层次结构

```

cin >> grade; //data "flows" is the direction of the arrows
              //to the right
  
```

注意,数据流读取操作符完全可以识别数据的类型。假定 `grade` 已经被正确声明,就没必要为流读取操作符指明更多类型的信息(某些 C 式的输入/输出偶尔会出现这样的情况)。

对象 `cout` 是预先定义的,它是 `ostream` 类的一个实例,和标准输出设备连在一起,通常是显示器。语句

```

cout << grade; //data "flows" in the direction of the arrows
              //to the left
  
```

使用的流插入操作符把整型变量 `grade`(假设 `grade` 已经声明为整型变量)从内存输出到标准输出设备。注意,数据流插入操作符也完全可以识别 `grade` 的类型。假定 `grade` 已被正确声明,就不必要为流插入操作符指明更多类型的信息。

对象 `cerr` 是预先定义的,它是 `ostream` 类的一个实例,和标准错误输出设备连在一起,输出到 `cerr` 对象并非是以缓存的方式的。也即每一个插入到 `cerr` 的流并同时执行其输出。这很适用于有错误发生时立刻提醒用户。

预先定义的对象 `clog` 是 `ostream` 类的一个实例,和标准错误输出设备连在一起,以缓存方式输出到 `clog` 对象。也即每一个插入到 `clog` 的流先是放在一个缓冲区一直到缓冲区满或是缓冲区被刷新。

C++ 文件处理机制使用 `ifstream` 类来操作文件输入、`ofstream` 类来处理文件输出、`fstream` 类来处理输入/输出操作。`ifstream` 类继承于 `istream`、`ofstream` 类继承于 `ostream`、`fstream` 类继承于 `iostream`。图 11.2 总结了这些输入/输出流相关类的各种继承关系。大多数的安装配置会提供更全面的输入/输出流相关的类层次,这比我们提到的类要多得多。我们这里所提到的类提供了程序员需要的主要的处理能力。有关文件的处理详情,参见类库参考手册。

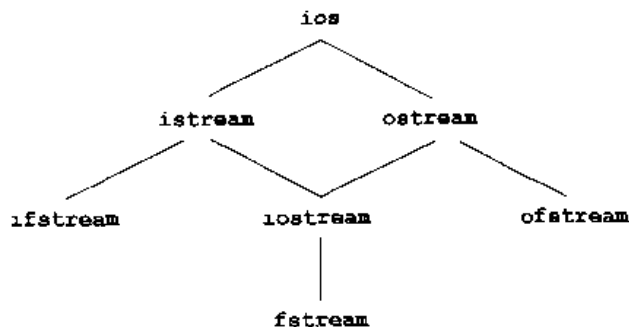


图 11.2 关键性文件处理类的部分 I/O 层次结构

## 11.3 输出流

C++ 的 `ostream` 类提供了格式化和无格式输出的功能。输出功能包括:用流插入操作符输出标准数据类型;用 `put` 成员函数输出字符;用 `write` 成员函数实现无格式输出(参见 11.5 节);以十进制、八进制、十六进制方式输出整型(参见 11.6.1 节);以各种精确方式输出浮点数(参见 11.6.2 节);强制输出带有小数点的浮点数(参见 11.7.2 节);以科学计数法和定点计数法表示浮点数(参见 11.7.6 节);在指定宽度的区域内对齐输出浮点数(参见 11.7.3 节);在区域内用特定字符填充(参见 11.7.4 节);以科学计数法和十六进制计数法输出大写字母(参见 11.7.7 节)。

### 11.3.1 流插入操作符

流的输出可以通过流插入操作符来实现(如重载操作符 `<<`)。操作符 `<<` 被重载之后输出内置类型数据项、字符串和指针值。11.9 节介绍了如何重载 `<<` 以输出用户自定义的类型。图 11.3 中的程序用一条流插入语句输出字符串。图 11.4 中的程序使用了多条流插入语句。图 11.4 的输出结果与图 11.3 的结果相同。

```
1 //Fig.11.3: fig11_03.cpp
2 //Outputting a string using stream insertion.
3 #include <iostream>
4
5 using std::cout;
6
7 int main()
8 {
9     cout << "Welcome to C++! \n";
10
11     return 0;
12 }
```

输出结果:

Welcome to C++!

图 11.3 用流插入操作符输出字符串

```
1 //Fig.11.4: fig11_04.cpp
2 //Outputting a string using two stream insertions.
3 #include <iostream>
4
5 using std::cout;
6
7 int main()
8 {
9     cout << "Welcome to ";
10    cout << "C++! \n";
11
12    return 0;
13 }
```

```
13 |
```

输出结果:

```
Welcom to C++!
```

图 11.4 用两条流插入操作符语句输出字符串

如图 11.5 所示,流操作符 `endl`(行结束)也可以实现 `\n`(换行符)的换行功能。流操作符 `endl` 会发送一个换行符,同时清空输出缓存(无论输出缓存是否已满都立即输出缓存中的数据)。语句

```
count << flush;
```

可用于清空输出缓存。流操作符将在 11.6 节会详细讨论。

表达式的输出如图 11.6 所示。

```
1 //Fig. 11.5; fig11_05.cpp
2 //Using the endl stream manipulator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     cout << "Welcome to ";
11     cout << "C++!";
12     cout << endl;    //end line stream manipulator
13
14     return 0;
15 }
```

输出结果:

```
Welcom to C++!
```

图 11.5 流操作符 `endl` 用法示例

```
1 //Fig. 11.6; fig11_06.cpp
2 //Outputting expression values.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     cout << "47 plus 53 is ";
11
12     //parentheses not needed; used for clarity
13     cout << ( 47 + 53 );    //expression
14     cout << endl;
15
16     return 0;
```

17 |

输出结果:

47 plus 53 is 100

图 11.6 输出表达式的值

**良好的编程习惯 11.2** 输出表达式时,将表达式用括号括起来以防止表达式和 << 操作符之间存在操作符优先级的问题。

### 11.3.2 流插入/流读取操作符的连续使用

重载后的操作符 << 和 >> 可以连续使用,如图 11.7 所示:

```
1 //Fig.11.7: fig11_07.cpp
2 //Cascading the overloaded << operator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 |
10     cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
11
12     return 0;
13 }
```

输出结果:

47 plus 53 is 100

图 11.7 重载操作符 &lt;&lt; 的连续使用

在图 11.7 中,多重流插入语句执行语句

```
(( (count << "47 plus 53 is ") << ( 47 + 53 )) << endl);
```

这种流插入操作符的连续使用是允许的,因为重载操作符 << 返回的是它左边操作对象(即 count)的引用。因此,最左边括号内的表达式

```
(count << "47 plus 53 is")
```

输出了指定的字符串并返回一个对 count 的引用。中间括号内的表达式可以解释为

```
(count << (47 + 53))
```

输出整数值 100 并返回一个对 count 的引用。最右边括号内的表达式可以解释为

```
count << endl
```

输出一个换行符,刷新 count 并返回对 count 的引用。最后返回的引用未被使用。

### 11.3.3 char \* 变量的输出

在 C 类型的输入/输出中,程序员必须提供数据类型信息。而 C++ 可以自动确定数据类型(这是对 C 的很好的改进)。但是有时在 C++ 中仍然有必要提供数据类型信息。例如,我们知道字符串是 char \* 型的。假定需要打印某个指针的值,即某个字符串的第一个字符在

内存中的地址。但是操作符 << 已经被重载用来打印以空字符串结尾的 char \* 型数据。这时只能将该指针强制转换为 void \* 型(void \* 型可以用来输出任何指针变量的地址)。图 11.8 中的示例以字符串和地址格式输出打印一个 char \* 变量。注意,地址的输出打印是十六进制(以 16 为基数)的数字格式。在 11.6.1 节、11.7.4 节、11.7.5 节和 11.7.7 节将具体介绍控制数值基数的方法。注意,图 11.8 中,程序的输出结果将随编译器的不同而不同。

```

1 //Fig. 11.8: fig11_08.cpp
2 //Printing the address stored in a char * variable
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     char *string = "test";
11
12     cout << "Value of string is: " << string
13         << "\nValue of static_cast< void * >( string ) is: "
14         << static_cast< void * >( string ) << endl;
15     return 0;
16 }
```

输出结果:

```

value of string is: test
Value of static_cast< void * >( string ) is: 0046C070
```

图 11.8 打印 char \* 变量中保存的地址

### 11.3.4 用成员函数 put 输出字符和 put 函数的连续调用

put 成员函数输出一个字符,如下所示

```
count.put('A');
```

将字符 A 显示在屏幕上。put 的连续调用如下

```
count.put('A').put('\n');
```

输出字符 A 并且输出一个换行符。与使用操作符 << 一样,上述语句之所以可以这样调用,是因为圆点操作符(.)从左至右的连结语句,put 成员函数返回 ostream 对象的引用。put 函数还可以随 ASCII 码值表达式一起调用,如 count.put(65),同样可以输出字符 A。

## 11.4 输入流

下面来看看输入流。输入流是通过流读取操作符(即重载后的操作符 >>)来完成的。通常情况下,该操作符会忽略输入流中的空白字符(如空格、制表位和换行符),稍后将介绍如何改变这种现象。遇到输入流的文件结束符时,流读取操作符会返回 0(false);反之,则返回接收该读取命令的对象的引用(即表达式 cin >> grade 中的 cin)。每个输入流都含有一系列状态位,用于控制流的状态(如格式化、出错状态设置等等)。如果输入了非法类型的数

据,流读取操作符就会设置流的 failbit 状态位;如果读取操作失败,就会设置 badbit 状态位。后面你会看到在输入/输出操作完成后,如何测试这些状态位。11.7 节和 11.8 节会详细讨论流的状态位。

### 11.4.1 流读取操作符

如图 11.9 所示,使用 cin 对象和重载后的流读取操作符 >> 输入两个整型数。注意流读取操作符也可以连续使用。

```
1 //Fig.11.9: fig11_09.cpp
2 //Calculating the sum of two integers input from the keyboard
3 //with cin and the stream-extraction operator.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 int main()
11 |
12     int x, y;
13
14     cout << "Enter two integers: ";
15     cin >> x >> y;
16     cout << "Sum of " << x << " and " << y << " is: "
17         << ( x + y ) << endl;
18
19     return 0;
20 |
```

输出结果:

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

图 11.9 利用 cin 和流读取操作符计算键盘输入的两个整数值和

操作符 << 和 >> 相对较高的优先级可能会出现问题。例如,图 11.10 所示的程序中,如果不在条件表达式中加括号,就无法得以正确编译。你可以亲自实践一下。

```
1 //Fig.11.10: fig11_10.cpp
2 //Avoiding a precedence problem between the stream-insertion
3 //operator and the conditional operator.
4 //Need parentheses around the conditional expression.
5 #include <iostream>
6
7 using std::cout;
8 using std::cin;
9 using std::endl;
10
11 int main()
```

```

12 |
13     int x, y;
14
15     cout << "Enter two integers: ";
16     cin >> x >> y;
17     cout << x << ( x == y ? " is" : " is not" )
18         << " equal to " << y << endl;
19
20     return 0;
21 |

```

输出结果:

```

Enter two integers: 7 5
7 is not equal to 5

```

```

Enter two integers: 8 8
8 is equal to 8

```

图 11.10 避免流插入操作符与条件操作符间出现优先级问题

**常见编程错误 11.1** 试图从 ostream 类(或其他只有输出流的类)的对象中读取数据。

**常见编程错误 11.2** 试图向 istream 类(或其他只有输入流的类)的对象写入数据。

**常见编程错误 11.3** 使用优先级相对较高的流插入操作符 << 或流读取操作符 >> 时,不用圆括号来强制执行正确的计算顺序。

输入一系列值的正确方法之一是:在 while 的循环条件中使用流读取操作符。遇到输入流的文件结束符时,流读取操作符会返回 0(false)。图 11.11 中的程序是用于查找一次考试的最高分。假设事先不知道考试成绩的个数,用户输入文件结束符以表明所有成绩已经输入。在 while 的条件里,(cin >> grade)遇到文件结束符会返回 0(即 false)。

```

1 //Fig. 11.11: fig11_11.cpp
2 //Stream-extraction operator returning false on end-of-file.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 |
11     int grade, highestGrade = -1;
12
13     cout << "Enter grade (enter end-of-file to end): ";
14     while( cin >> grade ) {
15         if ( grade > highestGrade )
16             highestGrade = grade;
17
18         cout << "Enter grade (enter end-of-file to end): ";
19     }

```



```

20
21     cout << "\n\nHighest grade is: " << highestGrade << endl;
22     return 0;
23 }

```

输出结果:

```

Enter grade (enter end-of-file to end) 67
Enter grade (enter end-of-file to end) 87
Enter grade (enter end-of-file to end) 73
Enter grade (enter end-of-file to end) 95
Enter grade (enter end-of-file to end) 34
Enter grade (enter end-of-file to end) 99
Enter grade (enter end-of-file to end) ^
Highest trade is: 99

```

图 11.11 流读取操作符遇到文件结束符时,返回 false

图 11.11 中的程序中, `cin >> grade` 被用作条件, 因为基类 `ios` (继承生成 `istream` 类) 提供了一个重载的类型强制转换的操作符, 将输入流转换为 `void *` 类型指针。如果在读取数值发生错误或遇到文件结束符时, 返回的指针值就是 0 (false)。编译器会隐式使用 `void *` 类型转换操作符。

#### 11.4.2 成员函数 `get` 和 `getline`

不带参数值成员函数 `get`, 从指定的流中读取一个字符 (即使是空格), 并返回该字符作为该函数调用的值。该版本的 `get` 函数遇到文件结束符时返回 EOF。

图 11.12 中的程序中, 使用了输入流 `cin` 的 `eof` 和 `get` 成员函数、输出流 `cout` 的 `put` 成员函数。首先打印 `cin.eof()` 的值, 即 false (输出为 0) 以表明 `cin` 没有遇到文件结束符。用户输入一行数据, 然后按回车键结束文件输入 (在 IBM 兼容机系统中, 文件结束是 `<Ctrl>-z`, UNIX 和 Macintosh 系统中则是 `<Ctrl>-d`)。程序读取每个字符并用成员函数 `put` 输出到 `cout`。但遇到文件结束操作符时, 跳出 `while` 循环, 并打印 `cin.eof()` 的值 (现在是 true) 1 表示 `cin` 收到了文件结束符。注意该程序使用的 `istream` 的成员函数 `get` 不带参数, 其返回值是输入的字符。

```

1 //Fig. 11.12; fig11_12.cpp
2 //Using member functions get, put and eof.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     char c;
12
13     cout << "Before input, cin.eof() is " << cin.eof()
14         << "\nEnter a sentence followed by end-of-file; \n";

```

```

15
16 while ( ( c = cin.get() ) != EOF )
17     cout.put( c );
18
19 cout << "\nEOF in this system is: " << c;
20 cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
21 return 0;
22 }

```

输出结果:

```

Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions?
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1

```

图 11.12 成员函数 get, put 和 eof 用法示例

以一个字符型引用作为参数的 get 成员函数读取输入流的下一个字符(包括空格),并且把它保存到字符型参数中。遇到文件结束符时,返回 0;反之,则返回调用该 get 成员函数的 istream 对象的引用。

get 函数还可以带 3 个参数:接收字符的字符型数组、数组大小和分隔符(默认为'\n'),从输入流中读取数据时,读取到比指定的最大字符数少一个字符时中止,或者读取到指定的分隔符时中止。为了使字符串数组(被程序用作缓冲区)中的输入字符串结束,空字符会被插入到字符数组中。分隔符不会被存储在数组中,但仍保留在输入流中(分隔符就是读取的下一个字符),所以除非分隔符从输入流中被刷新,否则紧接着的第二个 get 操作结果就是空行。图 11.13 中的程序比较了使用流读取操作符加 cin(读取字符直到遇到空字符为止)和 cin.get 来实现输入的区别。注意调用 cin.get 没有指定分隔符,使用默认的'\n'。

```

1 //Fig. 11.13: fig11_13.cpp
2 //Contrasting input of a string with cin and cin.get.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     const int SIZE = 80;
12     char buffer1[ SIZE ], buffer2[ SIZE ];
13
14     cout << "Enter a sentence;\n";
15     cin >> buffer1;
16     cout << "\nThe string read with cin was;\n"
17          << buffer1 << "\n\n";
18
19     cin.get( buffer2, SIZE );

```

```

20     cout << "The string read with cin.get was: \n"
21         << buffer2 << endl;
22
23     return 0;
24 |

```

输出结果:

```

Enter a sentence;
Contrasting string input with cin and cin.get

```

```

The string read with cin was;
Contrasting

```

```

The string read with cin.get was;
String input with cin and cin.get

```

图 11.13 使用流读取操作符 `cin` 输入和使用 `cin.get` 输入一个字符串之间的比较

成员函数 `getline` 的操作与带 3 个参数的 `get` 成员函数类似,读取一行字符串后在字符串数组中插入一个空格。不同的是,`getline` 会从输入流中删除分隔符(即读取字符并删除它),而不是把它放入数组。图 11.14 中的程序演示了如何用 `getline` 成员函数输入一行文本。

```

1 //Fig.11.14: fig11_14.cpp
2 //Character input with member function getline.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 |
11     const SIZE = 80;
12     char buffer[ SIZE ];
13
14     cout << "Enter a sentence: \n";
15     cin.getline( buffer, SIZE );
16
17     cout << " \nThe sentence entered is: \n" << buffer << endl;
18     return 0;
19 |

```

输出结果:

```

Enter a sentence;
using the getline member function

```

```

The sentence entered is:
Using the getline member function

```

图 11.14 用成员函数 `getline` 输入字符

### 11.4.3 istream 成员函数 peek, putback 和 ignore

成员函数 ignore 用于跳过指定数量的字符(默认数 1)或转到指定的分隔符(默认分隔符 eof,在读取文件时,该分隔符可使得 ignore 跳到文件末尾)中止输入。

成员函数 putback 的作用是把上一次从输入流中通过 get 取得的字符再放回该输入流中。对于应用程序需要扫描输入流以查找以特定字符开头的字段来说,这是非常有用的。输入一个字符,应用程序将该字符放回输入流,以保证输入的数据中包含该字符。

成员函数 peek 的功能是返回输入流的下一个字符,而不是从输入流中删除该字符。

### 11.4.4 输入/输出的类型安全性

C++ 提供输入/输出的类型安全性。操作符 << 和 >> 被重载以便能接受特定类型的数据。如果需要处理非法的数据类型时,将设置不同的错误标志,用户可以藉此测试输入/输出操作是否成功。从这个意义上说,程序是拥有控制权的。11.8 节将讨论这些错误状态标志。

## 11.5 成员函数 read, gcount 和 write 的无格式输入/输出

成员函数 read 和 write 专用于处理无格式的输入/输出。它们分别负责把一定量的字节输入内存的数组,以及从内存数组中输出字节。这些字节都未被格式化,均以原始数据的形式输入/输出。例如函数调用

```
char buffer[] = "HAPPY BIRTHDAY"
cout.write(buffer,10)
```

用于输出 buffer(缓存)中的前 10 个字节(包括会导致 cout 和 >> 输出时发生中断的空白字符)。因为字符串就是第一个字符的地址,所以函数调用

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ",10)
```

可以显示字母表的前 10 个字母。

成员函数 read 将指定数量的字符串输入字符型数组。如果读取的字符数量少于指定的数量,就会设置“failbit”错误状态位。我们很快会介绍如何判断是否已设置 failbit 设置(参见 11.8 节)。成员函数 gcount 可以统计最后一次输入操作读取的字符数。

图 11.15 中的程序演示了 istream 成员函数 read 和 gcount 及 ostream 成员函数 write。程序用 read 将 20 个字符(比数组长度更长的输入序列)输入字符型缓存,然后用 gcount 统计输入字符数量,用 write 输出缓存中的字符。

```
1 //Fig. 11.15; fig11_15.cpp
2 //Unformatted I/O with read, gcount and write.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
```

```
10 {
11     const int SIZE = 80;
12     char buffer[ SIZE ];
13
14     cout << "Enter a sentence;\n";
15     cin.read( buffer, 20 );
16     cout << "\nThe sentence entered was;\n";
17     cout.write( buffer, cin.gcount() );
18     cout << endl;
19     return 0;
20 }
```

输出结果:

```
Enter a sentence;
Using the read, write, and gcount member functions
The sentence entered was;
Using the read, write
```

图 11.15 成员函数 read、gcount 和 write 的无格式输入/输出

## 11.6 流操纵元

C++ 提供了许多流操纵元以完成格式化输出/输入,诸如设置域宽、设置精度、设置和清除格式化标志、设置字段填充字符、刷新流、在输入流中插入一行并刷新流、在输出流中插入空白字符、跳过输入流的空白字符等功能。下面将详细介绍这些功能。

### 11.6.1 整数流的基数:dec、oct、hex 和 setbase

整数通常被解释为十进制(基数为 10)的值。为了改变流中的整数基数,可以插入操作算法 hex 将基数设为十六进制(基数为 16),或插入操作算法 oct 将基数设置为八进制(基数为 8)。插入操作算法 dec 可以恢复为十进制基数。

流的技术也可以通过流操纵元 setbase 来改变,它带有一个整数参数,可选值有 10,8,16。因为 setbase 带有参数,所以也称为参数化流操纵元。使用 setbase 或其他的参数化操纵元需要在程序中包含头文件 <iomanip>。如果不显式改变流基数,流的基数就不会发生变化。图 11.16 中的程序演示了 hex,oct,dec 和 setbase 的用法。

```
1 //Fig. 11.16: fig11_16.cpp
2 //Using hex, oct, dec and setbase stream manipulators.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::hex;
```

```

12 using std::dec;
13 using std::oct;
14 using std::setbase;
15
16 int main()
17 {
18     int n;
19
20     cout << "Enter a decimal number: ";
21     cin >> n;
22
23     cout << n << " in hexadecimal is: "
24         << hex << n << '\n'
25         << dec << n << " in octal is: "
26         << oct << n << '\n'
27         << setbase( 10 ) << n << " in decimal is: "
28         << n << endl;
29
30     return 0;
31 }

```

输出结果:

```

Enter a decimal number: 20
20 in hexadecimal is: 15
20 in octal is: 24
20 in decimal is: 20

```

图 11.16 流操纵元 hex, oct, dec 和 setbase 用法示例

### 11.6.2 设置浮点数的精度: precision 和 setprecision

C++ 可以人为地控制浮点数的精度,即通过操作算法 setprecision 或 precision 成员函数来设置小数点右边的位数。精度一旦设置,就可用于以后所有的输出流操作,直至精度发生变化。图 11.17 中的程序用成员函数 precision 和流操纵元 setprecision 打印了 2 的平方根表,输出结果的精度从 0 不断变到 9。

```

1 //Fig.11.17: fig11_17.cpp
2 //Controlling precision of floating-point values
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::ios;
12 using std::setiosflags;
13 using std::setprecision;
14

```

```
15 #include <cmath>
16
17 int main()
18 {
19     double root2 = sqrt( 2.0 );
20     int places;
21
22     cout << setiosflags( ios::fixed)
23         << "Square root of 2 with precisions 0-9. \n"
24         << "Precision set by the "
25         << "precision member function;" << endl;
26
27     for ( places = 0; places <= 9; places ++ ) {
28         cout.precision( places );
29         cout << root2 << '\n';
30     }
31
32     cout << "\nPrecision set by the "
33         << "setprecision manipulator; \n";
34
35     for ( places = 0; places <= 9; places ++ )
36         cout << setprecision( places ) << root2 << '\n';
37
38     return 0;
39 }
```

输出结果:

Square root of 2 with precisions 0-9.  
Precision set by the precision member funtion;1

1.4  
1.41  
1.414  
1.4142  
1.41421  
1.414214  
4.4142136  
1.41421356  
1.414213562

Precision set by the setprecision manipulator:

1  
1.4  
1.41  
1.414  
1.4142  
1.41421  
1.414214  
1.4142136  
1.41421356  
1.414213562

图 11.17 控制浮点数的精度

### 11.6.3 设置域宽: setw 和 width

成员函数 `ios::width` 设置了当前域宽(即输入/输出的字符数)并且返回原先的设置。如果处理数据的域宽比设置的域宽小,会用填充字符来填充空位。如果数据的宽度比设置的宽度大,数据就不会被截断,系统将输出所有位数。

**常见编程错误 11.4** 宽度的设置仅适用于下一行的流插入或流读取,在该次操作完成之后,宽度被置回 0(即输出值按照所需的宽度来输出)不带参数的 `width` 函数返回当前的设置。认为域宽的设置适用于随后所有的输出,会导致逻辑错误。

**常见编程错误 11.5** 未对所处理的输出数据提供足够的宽度时,输出数据将按需要的域宽输出,这可能导致输出结果难以阅读。

图 11.18 中的程序演示了成员函数 `width` 在输入和输出上的用法。注意,输入一个字符数组后要加一个空白字符,所以读取的字符串域宽比设置的要小一位。遇到非前置空白字符时,流读取操作就会中止。流操纵元 `setw` 也可以用于设置域宽。注意,提示用户输入时,用户应该输入一行文本,并按回车键加上文件结束符(IBM PC 兼容机是 `<Ctrl>-z`;UNIX 和 Macintosh 是 `<Ctrl>-d`)。注意输入任何非字符型数组时,`width` 和 `setw` 会被忽略。

```

1  //fig11_18.cpp
2  //Demonstrating the width member function
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11     int w = 4;
12     char string[ 10 ];
13
14     cout << "Enter a sentence: \n";
15     cin.width( 5 );
16
17     while ( cin >> string ) {
18         cout.width( w++ );
19         cout << string << endl;
20         cin.width( 5 );
21     }
22
23     return 0;
24 }
```

输出结果:

```

Enter a sentence;
This is a test of the width member function
This
```



```

is
  a
test
  of
  the
  width
    h
  memb ~
    ex
  func
  tion

```

图 11.18 成员函数 width 用法示例

#### 11.6.4 用户自定义流操纵元

用户可以建立自己的流操纵元。图 11.19 中的程序演示了新的流操纵元 bell, ret, tab 和 endl 的建立和使用。用户也可以自行创建参数化流操纵元, 实现步骤可参考系统安装手册。

```

1 //Fig. 11.19: fig11_19.cpp
2 //Creating and testing user-defined, nonparameterized
3 //stream manipulators.
4 #include <iostream>
5
6 using std::ostream;
7 using std::cout;
8 using std::flush;
9
10 //bell manipulator (using escape sequence \a)
11 ostream& bell( ostream& output ) { return output << '\a'; }
12
13 //ret manipulator (using escape sequence \r)
14 ostream& ret( ostream& output ) { return output << '\r'; }
15
16 //tab manipulator (using escape sequence \t)
17 ostream& tab( ostream& output ) { return output << '\t'; }
18
19 //endLine manipulator (using escape sequence \n
20 //and the flush member function)
21 ostream& endLine( ostream& output )
22 {
23     return output << '\n' << flush;
24 }
25
26 int main()
27 {
28     cout << "Testing the tab manipulator;" << endLine
29         << 'a' << tab << 'b' << tab << 'c' << endLine
30         << "Testing the ret and bell manipulators;"
31         << endLine << ".....";

```

```

32     cout << bell;
33     cout << ret << "      " << endl;
34     return 0;
35 }

```

输出结果:

```

Testing the tab manipulator;
a      b      c
Testing the ret and bell manipulators;
——.....

```

图 11.19 建立和测试用户自定义的非参数化流操纵元

## 11.7 流格式状态

各种格式标志指定了即将在 I/O 流操作期间执行的格式类型。成员函数 `setf`, `unsetf` 和 `flags` 用于设置控制标志。

### 11.7.1 格式状态标志

图 11.20 中列出的格式状态标志在 `ios` 类中被定义为枚举型,具体细节参见后文描述。

| 格式状态标志                       | 说明                                                        |
|------------------------------|-----------------------------------------------------------|
| <code>ios::skipws</code>     | 跳过输入流的空白字符                                                |
| <code>ios::left</code>       | 左对齐域中的输出,必要时,在右边填充字符                                      |
| <code>ios::right</code>      | 右对齐域中的输出,必要时,在左边填充字符                                      |
| <code>ios::internal</code>   | 左对齐域中的数字符号,右对齐域中的数字值                                      |
| <code>ios::dec</code>        | 指定整数用十进制(基数 10)表示                                         |
| <code>ios::oct</code>        | 指定整数用八进制(基数 8)表示                                          |
| <code>ios::hex</code>        | 指定整数用十六进制(基数 16)表示                                        |
| <code>ios::showbase</code>   | 指定在数值前面输出进制(0 表示八进制;0x 或 0X 表示十六进制)                       |
| <code>ios::showpoint</code>  | 指定输出浮点数要带小数点。通常和 <code>ios::fixed</code> 一起使用保证小数点后面一定有位数 |
| <code>ios::uppercase</code>  | 指定输出十六进制整数时大写其中的字符(即 X 和 A 到 F),表示浮点数科学计数法的 e 也要大写        |
| <code>ios::showpos</code>    | 指定正数和负数前面要加 + 和 -                                         |
| <code>ios::scientific</code> | 指定浮点数输出采用科学计数法                                            |
| <code>ios::fixed</code>      | 指定浮点数输出采用定点符号,保证小数点后一定有位数                                 |

图 11.20 格式状态标志

成员函数 `flags`, `setf` 和 `unsetf` 用于控制这些状态标志,但是许多 C++ 程序员更喜欢使用流操纵元(参见 11.7.8 节)。程序员可以使用位或操作符(`|`)将不同的标志选项结合成一个 `long` 型的值(参见图 11.23)。调用成员函数 `flags` 并指定或操作选项,在流中设置在这些标志,并返回一个包括原先标志的选项 `long` 型。返回值通常要保存下来,以便调用 `flags` 根据保存值恢复原先的流选项设置。

函数 `flags` 指定一个代表所有标志的值。带有一个参数的函数 `setf` 则指定一个或多个或操作标志,并且把它们与现有的标志设置成“或”,形成新的格式状态。

参数化流操纵元 `setiosflags` 与成员函数 `setf` 具有同样功能。流操纵元 `resetiosflags` 与成员函数 `unsetf` 的功能相同。上述流操纵元在使用时都要添加头文件 `<iomanip>`。

标志 `skipws` 指示 `>>` 跳过输入流中的空白字符。`>>` 的默认行为是跳过空白字符,调用 `unsetf( ios::skipws )` 即可改变默认行为。流操纵元也可用于指定跳过空白字符。

### 11.7.2 追尾零和十进制小数点: `ios::showpoint`

设置 `showpoint` 标志强制输出浮点数的小数点和追尾零。如果不设置 `showpoint` 标志,浮点数 `79.0` 就会显示为 `79`;如果设置了 `showpoint`,就会显示为 `79.000000`(或追尾零的个数由当前的精度值决定)。图 11.21 中的程序演示了成员函数 `setf` 如何设置 `showpoint` 标志以控制追尾零的个数,同时打印浮点数的十进制小数点。

```
1 //Fig. 11.21: fig11_21.cpp
2 //Controlling the printing of trailing zeros and decimal
3 //points for floating-point values.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::ios;
12
13 #include <cmath>
14
15 int main()
16 {
17     cout << "Before setting the ios::showpoint flag\n"
18         << "9.9900 prints as: " << 9.9900
19         << "\n9.9000 prints as: " << 9.9000
20         << "\n9.0000 prints as: " << 9.0000
21         << "\n\nAfter setting the ios::showpoint flag\n";
22     cout.setf( ios::showpoint );
23     cout << "9.9900 prints as: " << 9.9900
24         << "\n9.9000 prints as: " << 9.9000
25         << "\n9.0000 prints as: " << 9.0000 << endl;
26     return 0;
27 }
```

输出结果:

```
Before setting the ios::showpoint flag
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9
```

```

After setting the ios::showpoint flag
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000

```

图 11.21 控制浮点值的追尾零和小数点的输出

### 11.7.3 对齐:ios::left,ios::right 和 ios::internal

left 和 right 标志分别用于从右填充字符以左对齐域和从左填充字符以右对齐域。用于填充的字符可以由成员函数 fill 或参数化的流操作符 setfill 指定(参见 11.7.4 节)。图 11.22 中的程序演示了如何用流操纵元 setw, setiosflags, resetiosflags 和成员函数 setf 与 unsetf 控制域中整型数据的左、右对齐。

```

1 //Fig.11.22: fig11_22.cpp
2 //Left - justification and right - justification.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::ios;
11 using std::setw;
12 using std::setiosflags;
13 using std::resetiosflags;
14
15 int main()
16 {
17     int x = 12345;
18
19     cout << "Default is right justified:\n"
20          << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
21          << "\nUse setf to set ios::left;\n" << setw(10);
22
23     cout.setf( ios::left, ios::adjustfield );
24     cout << x << "\nUse unsetf to restore default;\n";
25     cout.unsetf( ios::left );
26     cout << setw( 10 ) << x
27          << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
28          << "\nUse setiosflags to set ios::left;\n"
29          << setw( 10 ) << setiosflags( ios::left ) << x
30          << "\nUse resetiosflags to restore default;\n"
31          << setw( 10 ) << resetiosflags( ios::left )
32          << x << endl;
33     return 0;
34 }

```

输出结果:

Default is right justified;

12345

USING MEMBER FUNCTIONS

Use setf to set ios::left;

1234

Use unsetf to restore default;

12345

USING PARAMETERIZED STREAM MANIPULATORS

Use setiosflags to set ios::left;12345

Use resetiosflags to restore default;

12345

图 11.22 左对齐和右对齐

internal 标志表明域中一个数的符号位(设置 ios::showbase 标志时则为基数)要左对齐,数值要右对齐,中间的空白由填充字符填充。left,right 和 internal 标志都包含在静态数据成员 ios::adjusted 中。在设置 left,right 或 internal 对齐标志时,setf 的第二个参数必须是 ios::adjusted,因此 setf 只能设置 3 个对齐标志中的一个(它们是相互排斥的)。图 11.23 中的程序表明流操纵元 setiosflags 和 setw 用于指定内部空格,注意,用 ios::showpos 标志可以强制打印加号(+ )。

```
1 //Fig.11.23; fig11_23.cpp
2 //Printing an integer with internal spacing and
3 //forcing the plus sign.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::ios;
12 using std::setiosflags;
13 using std::setw;
14
15 int main()
16 {
17     cout << setiosflags( ios::internal | ios::showpos )
18         << setw( 10 ) << 123 << endl;
19     return 0;
20 }
```

输出结果:

+ 123

图 11.23 打印中间带有空格的整数并强制输出加号(+ )

### 11.7.4 填充:fill 和 setfill

成员函数 fill 指定了对齐域时用于填充的字符。如果没有指定具体值,默认使用空格来填充。函数 fill 返回原先的填充字符。流操纵元 setfill 也可以设置填充字符。图 11.24 中的程序演示了成员函数 fill 和流操纵元 setfill 是如何设置和恢复填充字符的。

```

1 //Fig. 11.24: fig11_24.cpp
2 //Using the fill member function and the setfill
3 //manipulator to change the padding character for
4 //fields larger than the values being printed.
5 #include <iostream>
6
7 using std::cout;
8 using std::endl;
9
10 #include <iomanip>
11
12 using std::ios;
13 using std::setw;
14 using std::hex;
15 using std::dec;
16 using std::setfill;
17
18 int main()
19 {
20     int x = 10000;
21
22     cout << x << " printed as int right and left justified\n"
23          << "and as hex with internal justification.\n"
24          << "Using the default pad character (space):\n";
25     cout.setf( ios::showbase );
26     cout << setw( 10 ) << x << '\n';
27     cout.setf( ios::left, ios::adjustfield );
28     cout << setw( 10 ) << x << '\n';
29     cout.setf( ios::internal, ios::adjustfield );
30     cout << setw( 10 ) << hex << x;
31
32     cout << "\n\nUsing various padding characters:\n";
33     cout.setf( ios::right, ios::adjustfield );
34     cout.fill( '*' );
35     cout << setw( 10 ) << dec << x << '\n';
36     cout.setf( ios::left, ios::adjustfield );
37     cout << setw( 10 ) << setfill( '%' ) << x << '\n';
38     cout.setf( ios::internal, ios::adjustfield );
39     cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
40     return 0;
41 }

```

输出结果:

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
```

```
    10000
10000
0x   2710
```

```
using various padding characters;
* * * * *10000
10000% % % % %
0x^^^^2710
```

图 11.24 用成员函数 `fill` 和流操纵元 `setfill` 改变实际域宽比域宽小的数据的填充字符

### 11.7.5 整数流的基数: `ios::dec`, `ios::oct`, `ios::hex` 和 `ios::showbase`

静态成员 `ios::basefield` (用法与 `setf` 的 `ios::adjustfield` 类似) 包含以下几个标志: `ios::oct`, `ios::hex` 和 `ios::dec` 标志位, 分别用于把整型数视为八进制、十六进制和十进制进行处理。如果不设置这些标志位, 流插入在默认情况下作为十进制数。流读取是按照整数被提供的方式来处理的 (即以零打头的整数按八进制数处理, 以 `0x` 或 `0X` 打头的按十六进制来处理, 其他所有的整数都是按十进制数来处理)。一旦为流设定了具体的基数, 流中的所有整数都按照该基数进行处理, 直到指定新的基数或程序结束。

`showbase` 的设定是用于强制输出整数值的基数。十进制数正常输出, 八进制数输出以 `0` 开头, 十六进制数输出要么以 `0x` 要么以 `0X` 开头 (具体由标志 `uppercase` 决定, 详情参见 11.7.7 节)。图 11.25 中的程序演示了使用 `showbase` 标志强制整数以十进制、八进制和十六进制格式打印。

```
1 //Fig.11.25: fig11_25.cpp
2 //Using the ios::showbase flag
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::ios;
11 using std::setiosflags;
12 using std::oct;
13 using std::hex;
14
15 int main()
16 {
17     int x = 100;
18
19     cout << setiosflags( ios::showbase )
20         << "Printing integers preceded by their base:\n"
21         << x << '\n'
22         << oct << x << '\n'
```

```

23         << hex << x << endl;
24     return 0;
25 |

```

输出结果:

```

Printing integers preceded by their base:
100
0144
0x64

```

图 11.25 ios::showbase 标志用法示例

### 11.7.6 浮点数和科学记数法:ios::scientific 和 ios::fixed

标志 ios::scientific 和 ios::fixed 包含在静态成员 ios::float 中,其用法与 setf 中的 ios::adjustfield 和 ios::basefield 类似。这些标志用于控制浮点数输出格式。标志 scientific 强制浮点数以科学记数法来输出。标志 fixed 强制浮点数按照定点格式显示,小数点后有指定位数(由成员函数 precision 指定)。若没有这些设置,就由浮点数的值决定输出格式。

调用 cout.setf(0,ios::floatfield) 恢复输出浮点数的默认格式。图 11.26 中的程序以定点格式和科学记数法输出浮点数,使用带有两个参数的 setf 成员函数,其中一个参数便是 ios::floatfield。科学记数法的典型格式会因为编译器的不同而不同。

```

1 //Fig.11.26: fig11_26.cpp
2 //Displaying floating-point values in system default,
3 //scientific, and fixed formats.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 int main()
11 {
12     double x = .001234567, y = 1.946e9;
13
14     cout << "Displayed in default format:\n"
15         << x << '\t' << y << '\n';
16     cout.setf( ios::scientific, ios::floatfield );
17     cout << "Displayed in scientific format:\n"
18         << x << '\t' << y << '\n';
19     cout.unsetf( ios::scientific );
20     cout << "Displayed in default format after unsetf:\n"
21         << x << '\t' << y << '\n';
22     cout.setf( ios::fixed, ios::floatfield );
23     cout << "Displayed in fixed format:\n"
24         << x << '\t' << y << endl;
25     return 0;
26 |

```

输出结果:



```

Displayed in default format:
0.00123457      1.946e+009
Displayed in scientific format:
1.234567e-003   1.946000e+009
Displayed in default format after unsetf:
0.001234567     1.946e+009
Displayed in fixed format:
0.001235        1946000000.000000

```

图 11.26 以系统默认格式、科学记数法和定点格式显示浮点数

### 11.7.7 大/小写的控制:ios::uppercase

输出十六进制的整数或科学计数法输出的浮点数时,标志 `ios::uppercase` 会强制输出大写的 X 或 E(参见图 11.27)。一旦设置了 `ios::uppercase` 标志,十六进制数中所有的字母都会以大写形式输出。

```

1 //Fig.11.27: fig11_27.cpp
2 //Using the ios::uppercase flag
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setiosflags;
11 using std::ios;
12 using std::hex;
13
14 int main()
15 |
16     cout << setiosflags( ios::uppercase )
17         << "Printing uppercase letters in scientific\n"
18         << "notation exponents and hexadecimal values:\n"
19         << 4.345e10 << '\n' << hex << 123456789 << endl;
20     return 0;
21 |

```

输出结果:

```

Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
75BCD15

```

图 11.27 `ios::uppercase` 标志用法示例

### 11.7.8 格式标志的设置和清除:flags, setiosflags 和 resetiosflags

不带参数的或员函数 `flags` 只返回格式标志的当前设置(long 型值)。带一个 long 型参

数的成员函数 `flags` 按照参数指定格式来设置标志,并返回原先的设置。`flags` 参数中未指定的任何格式都会被恢复。注意,系统不同,初始的格式设置也有所不同。图 11.28 中的程序演示了成员函数 `flags` 在设置新格式状态的同时,保留了原来的格式状态,最后再恢复原来的格式设置。

```

1  //Fig.11.28: fig11_28.cpp
2  //Demonstrating the flags member function.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::ios;
8
9
10 int main()
11 {
12     int i = 1000;
13     double d = 0.0947628;
14
15     cout << "The value of the flags variable is; "
16         << cout.flags()
17         << "\nPrint int and double in original format;\n"
18         << i << '\t' << d << "\n\n";
19     long originalFormat =
20         cout.flags( ios::oct | ios::scientific );
21     cout << "The value of the flags variable is; "
22         << cout.flags()
23         << "\nPrint int and double in a new format\n"
24         << "specified using the flags member function;\n"
25         << i << '\t' << d << "\n\n";
26     cout.flags( originalFormat );
27     cout << "The value of the flags variable is; "
28         << cout.flags()
29         << "\nPrint values in original format again;\n"
30         << i << '\t' << d << endl;
31     return 0;
32 }

```

输出结果:

```

The value of the flags variable is; 0
Print int and double in original format;
1000      0.0947628

```

```

The value of the flags variable is; 4040
Print int and double in a new format
specified using the flags member function:
1750      9.47628e-002

```

```

The value of the flags variable is; 0
Print values in original format again:

```

1000 0.0947628

图 11.28 成员函数 flags 用法示例

只要 long 型值出现在函数

```
Long previousFlagsettings =
    cout.setf(ios::showpoint | ios::showpos);
```

中,成员函数 setf 就会设置其参数所指定的格式标志,并恢复原来的标志设置值 long。带两个 long 型参数的成员函数 setf

```
cout.setf(ios::left, ios::adjustfield);
```

先清除 ios::adjustfield 位并且设置 ios::left 标志。该版本的 setf 函数用于与 ios::basefield (用 ios::dec, ios::oct 和 ios::hex 表示), ios::floatfield (用 ios::scientific 和 ios::fixed 表示) 和 ios::adjustfield (用 ios::left, ios::right 和 ios::internal 表示)。

成员函数 unsetf 清除指定的标志,并恢复清除前的标志值。

## 11.8 流错误状态

ios 类(用于输入/输出的 istream 类、ostream 类和 iostream 的基类)中的位可用于测试流的状态。

遇到文件结束符之后,将为输入流设置 eofbit 位。调用成员函数 eof 来确定是否已经遇到文件结束符。

流中发生格式错误时,虽然会设置 failbit,但字符不会丢失。成员函数 fail 判断流操作是否失败,这种错误通常可修复。

发生导致数据丢失的错误时,设置 badbit。成员函数 bad 判断流操作是否失败,这种严重错误通常不可修复。

如果 eofbit, failbit 或 badbit 都没有设置,则设置 goodbit。

如果函数 bad, fail 和 eof 全都返回 false,成员函数 good 就会返回 true。程序中应只对标记为“good”的流进行 I/O 操作。

成员函数 rdstate 返回流的错误状态。例如,函数调用 cult.rdstate 将返回流的状态,随后可用一条 switch 语句测试该状态,测试工作包括检查 ios::eofbit, ios::failbit 和 ios::goodbit。测试流状态的较好方法是使用成员函数 eof, bad, fail 和 good,使用这些函数不需要程序员熟知特定的状态位。

成员函数 clear 通常用于把一个流的状态恢复为“good”,从而可以对该流继续执行 I/O 操作。由于 clear 的默认参数位 ios::goodbit,所以语句

```
cin.clear();
```

清除 cin,并为流设置 goodbit。语句

```
cin.clear(ios::failbit)
```

实际上为流设置了 failbit。在用自定义类型对 cin 执行输入操作或遇到问题时,用户可能需要这么做。clear 一词似乎不适合用于此,但也不会错。

图 11.29 中的程序演示了成员函数 rdstate, eof, bad, good 和 clear 的用法。

```

1 //Fig.11.29; fig11_29.cpp
2 //Testing error states.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::cin;
8
9 int main()
10 |
11     int x;
12     cout << "Before a bad input operation:"
13         << "\ncin.rdstate(): " << cin.rdstate()
14         << "\n    cin.eof(): " << cin.eof()
15         << "\n    cin.fail(): " << cin.fail()
16         << "\n    cin.bad(): " << cin.bad()
17         << "\n    cin.good(): " << cin.good()
18         << "\n\nExpects an integer, but enter a character: ";
19     cin >> x;
20
21     cout << "\nAfter a bad input operation:"
22         << "\ncin.rdstate(): " << cin.rdstate()
23         << "\n    cin.eof(): " << cin.eof()
24         << "\n    cin.fail(): " << cin.fail()
25         << "\n    cin.bad(): " << cin.bad()
26         << "\n    cin.good(): " << cin.good() << "\n\n";
27
28     cin.clear();
29
30     cout << "After cin.clear()"
31         << "\ncin.fail(): " << cin.fail()
32         << "\ncin.good(): " << cin.good() << endl;
33     return 0;
34 |

```

#### 输出结果:

Before a bad input operation:

```

cin.rdstate(): 0
    cin.eof(): 0
    cin.fail(): 0
    cin.bad(): 0
    cin.good(): 1

```

Expects an integer, but enter a character: A

After a bad input operation:

```

cin.rdstate(): 2
    cin.eof(): 0
    cin.fail(): 0
    cin.bad(): 0

```

```
cin.good() 1
```

```
After cin.clear()
cin.fail(): 0
cin.good(): 1
```

图 11.29 测试错误状态

只要设置了 badbit 和 failbit 其中之一,成员函数 operator! 就返回 true。只要设置了 badbit 和 failbit 其中之一,成员函数 operator void \* 就会返回 false。这些函数可用于文件处理过程测试选择结构或循环结构条件的 true/false 情况。

## 11.9 把输出流连接到输入流

交互式的应用程序通常要用到输入的 istream 和输出的 ostream。屏幕上出现提示信息时,用户应键入相应的数据作出响应。显然,处理输入操作之前先要显示提示信息。在有输出缓存的情况下,只有当输出缓存已满、程序明确要求或程序结束时输出缓存的信息才会显示出来。C++ 提供了成员函数 tie 以同步 istream 和 ostream 的操作(即把两者联系起来)以保证输出在输入之前显示出来。调用语句

```
cin.tie(&cout);
```

把 cout(ostream 的对象)连接到 cin(istream 的对象)。事实上,这个特定的调用是多余的,因为 C++ 会自动执行这样的操作创建用户的标准输入/输出环境。但用户可以显式连接能匹配的 istream/ostream。解除从输出流到输入流的连接,可以使用调用

```
inputstream.tie(0)
```

## 11.10 小结

- I/O 操作对数据类型敏感。
- C++ 的 I/O 采用字节流的形式。流实际上是字节的序列。
- 系统的输入/输出机制以高效而可靠的方式实现数据在外部设备和内存之间的交互。
- C++ 提供了“低级”和“高级”的输入/输出功能。低级的 I/O 功能是在外部设备和内存之间传输一些字节。高级的 I/O 功能是将若干个字节组合成有意义的单位如整数、浮点数、字符、字符串和用户自定义类型。
- C++ 提供了无格式和格式化的 I/O 操作。无格式 I/O 传输效率高,但传输的数据是原始数据,使用较为麻烦。格式化的 I/O 处理的是有意义的单元,但会增加额外的处理时间,不适合处理大容量的数据。
- 大多数的 C++ 程序都包括 <iostream> 头文件,它声明了所有流 I/O 操作所需的信息。
- 头文件 <iomanip> 声明了参数化流操纵元,用于格式化输入/输出。
- <fstream> 头文件声明了文件处理所需的信息。

- `istream` 类支持流输入操作。
- `ostream` 类支持流输出操作。
- `istream` 类和 `ostream` 类均从基类 `ios` 继承而来。
- `iostream` 类从 `istream` 类和 `ostream` 类多重继承而来的。
- 重载后的左移位操作符 (`<<`) 表示流的输入,称为流插入操作符。
- 重载后的右移位操作符 (`>>`) 表示流的输出,成为流读取操作符。
- `cin` 是 `istream` 类的对象,连接标准输入设备,如键盘。
- `cout` 是 `ostream` 类的对象,连接标准输出设备,如屏幕。
- `ostream` 类对象 `cerr` 连接标准错误设备。输出到 `cerr` 属于非缓存输出的;每条错误信息会立即显示出来。
- 流操纵元 `endl` 会发送一个换行符,并同时清空输出缓存。
- C++ 编译器会自动地为输入/输出识别数据类型。
- 默认情况下,地址以十六进制的格式显示。
- 输出指针变量中的地址需要将指针强制转换成 `void *` 型
- 成员函数 `put` 用于输出单个字符。对 `put` 的调用可以连续使用。
- 使用流读取操作符 (`>>`) 实现流的输入。该操作符会自动的跳过输入流中的空白字符。
- 在使用操作符 `>>` 读取输入流时,遇到文件结束符,会返回 `false`。
- 使用流读取操作符时,遇到非法的输入时,会设置状态位 `failbit`;遇到操作失败时,会重新设置状态位 `badbit`。
- 成员函数 `ignore` 类似于带有 3 个参数的 `get` 函数。`getline` 会取消输入流中的分隔符,但不会将其保存在字符串内。
- 成员函数 `putback` 的功能是把最后一次用 `get` 从输入流中读取的字符放回输入流。
- 成员函数 `peek` 的功能是返回输入流的下一个字符,但并不从该流中读取(删除)该字符。
- C++ 提供了类型安全性的输入/输出。一旦操作符 `<<` 和 `>>` 操作时遇到非法的数据,就会设置各种错误标志,以使用户测试检查输入/输出是否成功。
- 成员函数 `read` 和 `write` 的功能是执行无格式的输入/输出。这两个函数分别把一定量的字节输入到指定的内存地址和从指定内存地址中输出。这些字节都是未经过格式化的,仅以原始数据形式输入/输出。
- 成员函数 `gcout` 的功能是返回最后一次 `read` 操作输入到流中的字符数。
- 成员函数 `read` 的功能是负责把指定量的字符输入到字符数组中,当输入的字符数少于指定的字符数,会设置状态位 `failbit`。
- 以下方法可以改变输出整数的基数:流操纵元 `hex` 用于把基数设为十六进制(即基数为 16);流操纵元 `oct` 用于把基数设为八进制(基数为 8);流操纵元 `dec` 将基数恢复为十进制。如果不明确修改基数设置,基数便不会发生变化。
- 整数输出的基数设置也可以通过参数化流操纵元 `setbase` 来实现。`setbase` 有一个整数参数是 10,8 或 16。

- 流操纵元 `setprecision`, `precision` 的功能是控制浮点数的精度。精度一旦设置,可适用于随后的输出操作,直到改变精度设置为止。不带参数的成员函数 `precision` 用于返回当前的精度值。
- 参数化流操纵元都需要在程序中包含有头文件 `<iomanip>`。
- 成员函数 `width` 用于设置域宽,并返回原先的域宽。如果输出的值长度比域宽小时,就会用填充字段来填充。域宽的设置只适用于下一次插入与读取;然后域宽隐式设为0(可以输出任意长度的序列)。长度大于域宽的值也可以完整输出。不带参数的函数 `width` 返回当前的域宽设置。流操纵元 `setw` 也可以用于设置域宽。
- 对于输入来说,流操纵元 `setw` 用于设置字符串长度的最大值。如果输入了其长度大于最大值的字符串,该字符串就会被分成若干个长度小于指定长度的字符串。
- 用户可自行建立流操纵元。
- 成员函数 `setf`, `unsetf` 和 `flags` 用于控制标志的状态。
- 标志 `skipw` 表明 `>>` 应该要跳过输入流上的空白字符。流操纵元 `ws` 还可以用于跳过输入流中的前导空白字符。
- 格式标志在 `ios` 类中被定义为枚举型。
- 成员函数 `flags`, `setf` 和 `unsetf` 用于控制这些状态标志,但是许多C++程序员更喜欢使用流操纵元。程序员可以使用位或操作符(`|`),将不同的标志选项结合成一个 `long` 型的值。调用成员函数 `flags` 并指定或操作选项,在流中设置在这些标志,并返回一个包括原先标志的选项 `long` 型。返回值通常要保存下来,以便调用 `flags` 根据保存值恢复原先的流选项设置。
- 函数 `flags` 指定一个代表所有标志的值。带有一个参数的函数 `setf` 则指定一个或多个或操作标志,并且把它们与现有的标志设置成“或”,形成新的格式状态。
- 设置 `showpoint` 标志强制输出浮点数的小数点和由精度指定的有效数字的个数。
- `left` 和 `right` 标志分别用于从右填充字符来左对齐域和从左填充字符来右对齐域。
- `internal` 标志指示域中的一个数的符号位(设置了 `ios::showbase` 标志时,则为基数)要左对齐,数值要右对齐,中间的空白由填充字符填充。
- `ios::adjustfield` 中包含了标志 `left`, `right` 和 `internal`。
- 成员函数 `fill` 与 `left`, `right` 和 `internal` 一起使用,指定了对齐域时用于填充的字符(默认是空格),返回原来填充的字符。流操纵元 `setfill` 也可以用于设置填充字符。
- 静态成员 `ios::basefield`(用法类似于 `setf` 的 `ios::adjustfield`)包括了以下几个标志:  
`ios::oct`, `ios::hex` 和 `ios::dec` 标志位,分别用于指定将整型数作为八进制、十六进制和十进制处理。如果不设置这些标志位,流插入符会在默认情况下将其作为十进制数。流读取符是按照整数方式进行处理。
- 设置 `showbase` 标志以强制输出整数值的基数。
- 标志 `ios::scientific` 和 `ios::fixed` 包含在静态成员 `ios::float` 中。标志 `scientific` 强制浮点数以科学记数法输出。标志 `fixed` 强制浮点数按照定点格式显示,小数点后的指定位数(由成员函数 `precision` 指定)。
- 调用 `cout.setf(0, ios::floatfield)` 恢复输出浮点数的默认格式。

- 输出十六进制的整数或科学计数法输出的浮点数时标志 `ios::uppercase` 强制输出大写的 X 或 E (参见图 11.27)。一旦设置了 `ios::uppercase` 标志,十六进制数中所有的字母都会以大写形式输出。
- 不带参数的成员函数 `flags` 只返回格式标志的当前设置(long 型值)。带一个 long 型参数的成员函数 `flags` 按照参数指定格式来设置标志,并返回原先的设置。
- 成员函数 `setf` 设置参数所指定的格式标志,并返回 long 类型原先的标志设置值。
- 成员函数 `setf(long setBits 和 long resetBits)` 用于清除 `resetBits` 中的位,并设置 `setBits` 中的位。
- 成员函数 `unsetf` 清除指定的标志并恢复清除前的标志值。
- 带参数的流操纵元 `setiosflags` 与成员函数 `flags` 执行同样的功能。
- 带参数的流操纵元 `resetiosflag` 与成员函数 `unsetf` 执行同样的功能。
- `ios` 类中的位可用于测试流的状态。
- 遇到文件结束符之后,为输入流设置 `eofbit` 位。
- 调用成员函数 `eof` 用于确定是否已经遇到文件结束符。
- 当流中发生格式错误时,虽然会设置 `failbit`,但是字符不会丢失。
- 成员函数 `fail` 判断流操作是否失败,这种错误通常可修复。
- 出现导致数据丢失的错误时,设置 `badbit`。成员函数 `bad` 判断流操作是否失败,这种严重错误通常不可修复。
- 如果函数 `bad`, `fail` 和 `eof` 全都返回 `false`,成员函数 `good` 就会返回 `true`。程序中应只对标记为“good”的流进行 I/O 操作。
- 成员函数 `rdstate` 返回流的错误的状态。
- 成员函数 `clear` 通常用于把一个流的状态恢复为“good”,从而可以对该流继续执行 I/O 操作。
- C++ 提供了成员函数 `tie` 以同步 `istream` 和 `ostream` 的操作(即把两者联系起来)以保证输出在输入之前显示。

## 本章术语

< iomanip > standard header file

标准头文件 < iomanip >

bad member function 成员函数 bad

clear member function 成员函数 clear

dec stream manipulator 流操纵元 dec

default fill character(space) 默认填充字符(空格)

default precision 默认精度

end-of-file 文件结束符

eof member function 成员函数 eof

fail member function 成员函数 fail

field width 域宽

fill character 填充字符

fill member function 成员函数 fill

flags member function 成员函数 flags

flush member function 成员函数 flush

flush stream manipulator 流操纵元 flush

format flags 格式标志

format states 格式状态

formatted I/O 格式化 I/O

fstream class ifstream 类

gcount member function 成员函数 gcount

get member function 成员函数 get

getline stream manipulator 流操纵元 getline

good member function 成员函数 good

hex stream manipulator 流操纵元 hex

ifstream class ifstream 类



ignore member function 成员函数 ignore  
 in-memory formatting 内存格式化  
 ios class ios 类  
 iostream class iostream 类  
 istream class istream 类  
 leading 0 (octal 0 开头(八进制))  
 leading 0x 或 0X (hexadecimal)  
 0x 或 0X 开头(十六进制)  
 left-justified 左对齐  
 oct stream manipulator 流操纵元 oct  
 ofstream class ofstream 类  
 operator ! member function 成员函数 operator!  
 operator void \* member function  
 成员函数 operator void \*  
 ostream class ostream 类  
 padding 填充  
 parameterized stream manipulator 参数化流操纵元  
 peek member function 成员函数 peek  
 precision member function 成员函数  
 predefined streams 预先定义的流  
 put member function 成员函数 put  
 putback member function 成员函数 putback  
 rdbuf member function 成员函数 rdbuf  
 read member function 成员函数 read  
 resetiosflags stream manipulator  
 流操纵元 resetiosflags  
 right-justified 右对齐  
 setbase stream manipulator 流操纵元 setbase  
 setf member function 成员函数 setf  
 setfill stream manipulator 流操纵元 setfill  
 setiosflags stream manipulator 流操纵元 setiosflags  
 setprecision stream manipulator 流操纵元 setprecision  
 setw stream manipulator 流操纵元 setw  
 stream input 输入流  
 stream insertion operator( << ) 流插入操作符( << )  
 stream manipulator 流操纵元  
 stream output 输出流  
 stream-extraction operator( >> )  
 流读取操作符( >> )  
 tie member function 成员函数 tie  
 type-safe I/O 类型安全性 I/O  
 unformatted I/O 无格式 I/O  
 unself member function 成员函数 unself  
 uppercase 大写  
 user-defined streams 用户自定义流  
 whitespace character 空白字符  
 width 宽度  
 write member function 成员函数 write  
 ws member function 成员函数 ws

## 常见编程错误

- 11.1 试图从 ostream 类(或其他只有输出流的类)的对象中读取数据。
- 11.2 试图向 istream 类(或其他只有输入流的类)的对象写入数据。
- 11.3 使用优先级相对较高的流插入操作符 << 或流读取操作符 >> 时,不用圆括号来强制执行正确的计算顺序。
- 11.4 域宽的设置仅适用于下一行的流插入或流读取,在该次操作完或之后,域宽被置回 0 (即输出值按照所需的域宽输出),不带参数的 width 函数返回当前的设置。认为域宽的设置适用于之后所有的输出,就会导致逻辑错误。
- 11.5 未对所处理的输出数据提供足够的域宽,输出数据将按需要的域宽输出,这可能导致输出结果难以阅读。

## 良好编程习惯

- 11.1 尽管 C 语言风格的 I/O 实际可用于 C++,但在 C++ 编程中尽量只用 C++ 风格的 I/O。
- 11.2 输出表达式时,将表达式用括号括起来以防止表达式和 << 操作符之间存在操作符优先级的问題。

## 性能提示

11.1 对于大容量文件的处理,使用无格式的输入/输出可获得最佳性能。

## 软件工程知识

11.1 C++ 类型的输入/输出具有类型安全性。

11.2 C++ 允许采用同样的方式处理用户自定义类型和输入/输出预先定义的类型。这样可以提高软件开发速度,尤其是提高软件的可重用性。

## 自测题

### 11.1 填空题:

- a) 重载后的流操作符常被定义为类的\_\_\_\_\_函数。
- b) 可以设置的用于格式化对齐的位包括\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- c) C++ 的输入/输出基于字节的\_\_\_\_\_形式而实现的。
- d) 参数化流操纵元\_\_\_\_\_和\_\_\_\_\_被用来设置和恢复格式状态标志。
- e) 大多数C++ 程序通常都包括\_\_\_\_\_头文件,它包含了所有输入/输出操作所需要的信息。
- f) 成员函数\_\_\_\_\_和\_\_\_\_\_用来设置和恢复格式状态标志。
- g) 头文件\_\_\_\_\_包含了执行内存格式化所要声明的信息。
- h) 要使用参数化的流操纵元,必须包含有头文件\_\_\_\_\_。
- i) 头文件\_\_\_\_\_包含了用户控制的文件处理所需的信息。
- j) 流操纵元\_\_\_\_\_在输出流中插入换行符,并刷新输出流。
- k) 在 C 类型和C++ 类型混合的 I/O 程序里要使用头文件\_\_\_\_\_。
- l) ostream 类的成员函数\_\_\_\_\_用于执行无格式输出。
- m) \_\_\_\_\_类支持输入操作。
- n) 类标准错误流的输出指向对象\_\_\_\_\_或对象\_\_\_\_\_。
- o) \_\_\_\_\_类支持输出操作。
- p) 流插入操作符是\_\_\_\_\_。
- q) 与标准系统外部设备有关的 4 类对象是:\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- r) 流读取操作符是\_\_\_\_\_。
- s) 流操纵元\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_分别指定了整数的输出以十进制、八进制和十六进制的格式显示。
- t) 用于显示浮点数的默认精度值是\_\_\_\_\_。
- u) 一旦标志设置\_\_\_\_\_设置,就可以使显示的正数前带上加号。

### 11.2 判断正误,如果不正确,请说明原因。

- a) 带有 long 型参数的流成员函数 flags() 用于根据所带参数设置标志状态,并返回原来的设置。
- b) 流插入操作符 << 和流读取操作符 >> 被重载以处理所有的标准数据类型——包括字符和内存地址(仅对流插入)——和全部用户自定义类型。

- c) 不带参数的流成员函数 `flags()` 用于恢复标志状态变量的所有标志位。
- d) 流读取操作符 `>>` 重载了一个操作符函数,以 `istream` 引用和用户自定义类型的引用作为参数,返回一个 `istream` 引用。
- e) 流操作符 `ws` 跳过输入流的前导空白字符。
- f) 流插入操作符 `<<` 重载了一个操作符函数,以 `istream` 引用和用户自定义类型的引用作为参数,返回一个 `istream` 引用。
- g) 用流读取操作符 `>>` 输入可以跳过输入流的前导空白字符。
- h) 输入/输出功能是 C++ 标准库中的一部分。
- i) 流成员函数 `rdstate()` 返回流当前的状态。
- j) 流操作符 `cout` 通常与显示屏有关。
- k) 如果函数 `bad`, `fail` 和 `eof` 全都返回 `false`,则成员函数 `good()` 返回 `true`。
- l) 流操作符 `cin` 通常与显示屏有关。
- m) 在流操作时发生不可恢复的错误时,成员函数 `bad` 返回 `true`。
- n) `cerr` 的输出是非缓存的,`clog` 的输出是缓存的。
- o) 设置了 `ios::showpoint` 后,浮点数的值将强制以默认精度 6 位小数的方式输出(前提是精度值没有被改变,其时会用具体精度输出浮点值)。
- p) `ostream` 类成员函数 `put` 输出指定数量的字符。
- q) 流操纵元 `dec`, `oct` 和 `hex` 只对下一次输出操作产生影响。
- r) 在输出时,内存地址以默认的方式显示为 `long` 型整数。

### 11.3 针对下列各项任务,编写语句。

- a) 输出“Enter your name”字符串。
- b) 设置一个标志,使科学记数法的指数以及十六进制数中的字符以大写形式输出。
- c) 输出 `char *` 型变量的 `string` 的地址。
- d) 设置一个标志,以科学记数法输出浮点数。
- e) 输出 `int *` 型的变量 `integerPtr` 的地址。
- f) 设置标志,使得在输出整数时,八进制数和十六进制数显示其基数。
- g) 输出 `float *` 类型变量 `floatPtr` 所指向的地址。
- h) 当所设的域宽长度大于输出数据所需宽度时,用流成员函数将用于填充的字符设置为 `*`,再写一条语句。
- i) 用流操纵元来实现。用 `ostream` 类的函数 `put` 写一条语句输出字符 `'O'` 和 `'K'`。
- j) 从输出流中获取下一个字符,但不提取它。
- k) 利用 `istream` 类的成员函数 `get` 以两种方式向 `char` 型的变量 `c` 输入字符。
- l) 输入并删除输入流中的 6 个字符。
- m) 利用 `istream` 类的成员函数 `read` 向 `char` 型数组 `line` 输入 50 个字符。
- n) 向字符串数组 `name` 中读入 10 个字符。遇到分隔符 `!` 时结束。不要删除输入流中的分隔符。再写一条语句完成上述功能,但要删除输入流中的分隔符。
- o) 利用 `istream` 类的成员函数 `gcount` 统计字符型数组 `line` 中的字符数,`line` 中的字符通过调用 `istream` 类的 `read` 函数输入,然后根据统计的字符数,用 `ostream` 类的成员函数 `write` 输出。
- p) 分别用成员函数和流操纵元刷新输出流。

- q) 输出下列值:124,18.376,'z',100 000 和 'string'。
- r) 利用成员函数显示当前的精度。
- s) 输入一个整数给 int 变量 months,输入一个浮点数给 float 型变量 percentageRate。
- t) 利用流操纵元输出 1.92,1.925,1.925 8,精度是 3 位小数。
- u) 利用流操纵元分别输出整数 100 的十进制值、八进值和十六进制值。
- v) 利用同一个流操纵元,通过改变基数来输出整数 100 对应的十进制值、八进值和十六进制值。右对齐且域宽为 10,输出 1 234。
- w) 读取字符给字符数组 line,直到遇到分隔符 'z' 或读取了 20 个字符时停止操作。该语句不从输入流中读取分隔符。
- x) 利用整数变量 x 和 y 指定域宽和精度以显示 double 型值 87.457 3。
- y) 按域宽 x、精度 y(x,y 为整型变量)输出 double 类型的值 87.457 3。

#### 11.4 指出并改正下列语句的错误。

- a) `cout << "Value of x <= y is : " << x <= y`
- b) 下面的语句要输出字符 'c' 的整数值:`cout << 'c';`
- c) `cout << " "A string in quotes" "`;

#### 11.5 写出下面语句的输出结果。

- a) `cout << "12345" << endl;`  
`cout.width( 5);`  
`cout.fill( '*' );`  
`cout <<123 << endl << 123;`
- b) `cout << setw( 10 ) << setfill( '$' ) << 1000;`
- c) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`
- d) `cout << setiosflags( ios::showbase ) << oct < 99`  
`<< endl << hex << 99;`
- e) `cout << 10000 << endl`  
`<< setiosflags( ios::showpos ) << 100000;`
- f) `cout << setw( 10 ) << setprecision( 2 ) <<`  
`<< setiosflags( ios::scientific ) <<444.93738`

### 自测题答案

- 11.1 a) 友元 b) `ios::left`, `ios::right` 和 `ios::internal`. c) 流 d) `setiosflags`, `resetiosflags`.  
 e) `iostream` f) `setf`, `unsetf`. g) `stringstream` h) `iosmanip` i) `ostream` j) `endl`  
 k) `std::ostream` l) `write` m) `istream` n) 或 `clog` o) `ostream` p) `<<` q) `cin`, `cout`, `cerr` 和 `clog` r) `>>` s) `oct`, `hex` 和 `dec` t) 精度 6 位小数 u) `ios::showpos`
- 11.2 a) 正确。  
 b) 错误。流插入操作符和流读取操作符不能重载用于用户自定义类型。程序员必须为每个用户自定义类型提供重载该操作符的操作符函数。  
 c) 错误。不带参数的流成员函数 `flags()` 值返回当前状态变量 `flags` 的状态值  
 d) 正确。  
 e) 正确。  
 f) 错误。要使用重载的流插入操作符,重载操作符函数必须包括 `ostream` 对象的引用

和用户自定义类型对象的引用为参数,同时返回 ostream 对象的引用。

- g) 正确。除非 ios::skipws 未设置标志位。
- h) 错误。C++ 的输入/输出功能是 C++ 标准库的一部分。C++ 语言本身并不包括输入、输出和文件处理功能。
- i) 正确。
- j) 正确。
- k) 正确。
- l) 错误。cin 流通常与计算机键盘有关。
- m) 正确。
- n) 正确。
- o) 正确。
- p) 错误。ostream 类的成员函数 put 用于输出单个字符。
- q) 错误。流操纵元 dec, oct 和 hex 用于设置输出整数的基数,直到下一次设置时或者程序中止时才会改变。
- r) 错误。默认情况下,内存地址以十六进制形式输出。要想以 long 型形式输出地址,必须将其类型强制转换为 long。

- 11.3
- a) `cout << "Enter your name: ";`
  - b) `cout.setf(ios::uppercase);`
  - c) `cout << (void *) string;`
  - d) `cout.setf(ios::scientific, ios::floatfield);`
  - e) `cout << integerPtr;`
  - f) `cout << setiosflags(ios::showbase);`
  - g) `cout << *floatPtr;`
  - h) `cout.fill( '*');`  
`cout << setfill( '*');`
  - i) `cout.put( 'o').put( 'k');`
  - j) `cin.peek();`
  - k) `c = cin.get();`  
`cin.get( c);`
  - l) `cin.ignore( 6 );`
  - m) `cin.read( line, 50 );`
  - n) `cin.get( name, 10, '.' );`  
`cin.getline( name, 10, '.' );`
  - o) `cout.write( line, cin.gcount() );`
  - p) `cout.flush();`  
`cout << flush;`
  - q) `cout << 124 << 18.376 << 'z' << 1000000 << "String";`
  - r) `cout << cout.precision();`
  - s) `cin >> months >> percentageRate;`
  - t) `cout << setprecision( 3 ) << 1.92 << '\t '`  
`<< 1.925 << '\t ' << 1.9258`

```

u) cout << oct << 100 << hex << 100 << dec << 100
v) cout << 100 << setbase( 8 ) << 100 << setbase( 16 ) << 100;
w) cout << setw( 10 ) << 1234;
x) cin.get( line, 20, 'z' );
y) cout << setw( x ) << setprecision( y ) << 87.4573;

```

11.4 a) 错误,操作符 << 的优先级高于 <= 的优先级,语句求值不正确,会导致编译器报错。  
改正:在表达式 `x <= y` 上加上括号。当任何一个表达式使用的操作符优先级低于操作符 <<, 而且该表达式没有加括号时,常会发生这种问题。

b) 错误,C++ 不能像 C 语句那样直接输出字符的 ASCII 码。

改正:要输出字符的 ASCII 必须先要计算出它的 ASCII 值,如下所示,把字符转换成整型值输出

```
cout << int( 'c' )
```

c) 错误,除非使用转义序列,否则不能输出引号。

改正:以下列方式打印字符

```

cout << ' ' << "A string in quotes" << ' ' ;
cout << "\A string in quotes \" ;

```

## 练习题

11.6 针对下述各条要求,编写 C++ 语句。

- 以左对齐方式输出整数 40 000,域宽为 15。
- 把一个字符串读入字符型数组变量 `state`。
- 打印有符号数 200 和无符号数 200。
- 将十进制整数 100 以 0x 开头的十六进制格式输出。
- 把字符读入数组 `s`,直到遇到字符 'p' 或读到的字符个数达到限定值 10 时才停止读取操作。同时从输入流中读取并删除分隔符。
- 以前导 0 格式打印 1.234,域宽为 9。
- 从标准输入流中读取字符串 "characters",将其保存在字符数组 `s` 中。读取过程中去掉双引号,读取字符个数的最大限定值为 50(包括空字符)。

11.7 编写一个程序,测试十进制、八进制、十六进制格式整数值的输入,分别按 3 种不同的基数输出。测试数据为 10,010 和 0x10。

11.8 编写一个程序,用强制类型转换操作符把指针值转换为各种整数数据类型后,打印出指针值。哪种数据类型会打印出奇怪的值? 哪种数据类型会产生错误?

11.9 编写一个程序,分别用不同的域宽打印整数 12 345 和浮点数 1.234 5。域宽小于数值实际需要的域宽时,会发生什么情况?

11.10 编写一个程序,将 100.453 627 取整为最接近的个位、十分位、百分位、千分位和万分位,并打印出结果。

11.11 编写一个程序,从键盘输入一个字符串,判断字符串的长度,然后以字符串长度的两倍作为域宽打印该字符串。

11.12 编写一个程序,将华氏温度 0 度 ~ 212 度转换为浮点型的摄氏温度,浮点数精度为 3。

转换公式为

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

输出用两个右对齐列,摄氏温度前要加正负号。

- 11.13 不同的编程语言中,字符串有时用单引号('),有时则用双引号(")。编写3个程序,读取3个不同的字符串 suzy, "suzy" 和 'suzy'。看单引号和双引号是被忽略了,还是被视为字符串的一部分读取了。
- 11.14 图 8.3 中的程序为 `phonenumbers` 类一起的输入和输出对象重载了流读取操作符和流插入操作符。重新编写流读取操作符,使它能够检测下面的输入错误。注意,函数 `operator >>` 的代码全部需要重写。
- 将一个完整的电话号码输入一个字符数组,检测输入的字符总数,例如电话号码 (800)555-1212 的字符总数为 14。如果输入错误,用流成员函数 `clear` 设置 `ios::failbit` 位。
  - 电话号码中的区号和局号不能以 0 或 1 开头。检测区号和局号的第一位,判断它们是 0 还是 1,若是,就用流成员函数 `clear` 设置 `ios::failbit` 位。
  - 区号的中间一位是 0 或 1,检测中间位,判断是 0 还是 1,如果输入错误,就用流成员函数 `clear` 设置 `ios::failbit` 位。如果正确,即 `ios::failbit` 位为 0,则把电话号码的 3 部分分别复制到对象 `phonenumbers` 的成员 `areacode`, `exchange` 和 `line` 中。在主程序里,如果设置 `ios::failbit` 位,程序就会打印出一条出错信息,并在不打印电话号码的情况下结束运行。
- 11.15 编写完成下列要求的程序:
- 创建用户自定义 `Point` 类,该类包含 `private` 整数数据成员 `xCoordinate` 和 `yCoordinate`,并在类中将重载的流插入和流读取操作符函数声明为其友元。
  - 定义流插入和流读取操作符函数。流读取操作符函数判断输入的数据是否合法,如果是非法数据,则设置 `ios::failbit` 位,表明输入不正确。发生错误后,流插入操作符函数将不显示 `point` 的对象(点)的坐标信息。
  - 编写函数 `main`,用重载的流插入和流读取操作符函数测试自定义 `Point` 类的输入和输出。
- 11.16 编写完成下列要求的程序:
- 创建用户自定义复数类 `Complex`,类中包括 `private` 整数数据成员 `real` 和 `imaginary`,并将重载的流插入和流读取操作符函数声明为其友元。
  - 定义流插入和流读取操作符函数。流读取操作符函数判断输入的数据是否合法,如果是非法数据,则设置 `ios::failbit` 位以表明输入不正确。必须按以下格式输入:  

$$3 + 8i$$
  - 数据可为整数或负数,还可以只为其中一个数赋值。未给出的数值,由相应的数据成员设置为 0。发生输入错误时,流插入操作符不显示该负数值。输出格式与上述的输入格式相同,为负数的虚部要打印出负号。
  - 编写函数 `main`,用重载的流插入和流读取操作符测试自定义 `Complex` 类的输入和输出。

- 11.17 用 for 结构为 ASCII 字符集中 33 ~ 126 之间的 ASCII 字符打印一张 ASCII 码表。要求输出十进制、八进制、十六进制值和 ASCII 码值,并在程序中使用流操纵元 `dec`, `oct` 和 `hex`。
- 11.18 编写一个程序,用成员函数 `getline` 和带 3 个参数的成员函数 `get` 输入带有空字符的字符串。`get` 函数不读取分隔符,分隔符仍保留在输入流中,并让 `getline` 从输入流中读取并删除分隔符。把未读取的字符留在输入流中会发生什么情况?
- 11.19 编写一个程序,建立用户自定义操纵算法 `skipwhite` 以跳过输入流中的前导空白字符。该操纵算法利用 `<cctype>` 函数库中的 `isspace` 函数测试输入的字符是否是空白字符。要求用 `istream` 的成员函数 `get` 读取每个字符,遇到一个非空白字符时,流操纵元 `skipwhite` 会把该字符放回输入流,返回对 `istream` 对象的引用。
- 建立一个 `main` 函数,测试自定义流操纵元(在 `main` 函数中清除 `ios::skipws` 标志位,以保证流读取运算不会自动跳过空白字符)。然后输入以空白字符开头的字符,测试流操纵元,读取完毕之后,输出所读取的字符以确定的确没有输入空白字符。



# 第 12 章 模 板

## 学习目标

- 能用函数模板创建相关(重载)函数组
- 能区分函数模板与模板函数
- 能用类模板创建相关类型组
- 能区分类模板与模板类
- 理解如何重载模板函数
- 理解模板、友元、继承与静态成员之间的关系

## 12.1 简介

本章介绍C++ 最强大的功能之一:模板。模板使我们可用单个代码段来指定一组相关(重载)函数(称为模板函数)或一组相关类(称为模板类)。

我们可以为数组排序函数编写一个函数模板,然后让C++ 自动生成多个模板函数,并对 int 数组、float 数组和字符串数组等进行排序。

第 3 章中介绍了函数模板。这里将没有阅读第 3 章的读者提供一些基础知识与示例。

可以为堆栈类编写一个类模板,然后让C++ 自动生成如 int, float 和 string 堆栈类的类模板。

注意函数模板与模板函数的区别:函数模板和类模板如同具有各种形状的模板,模板函数和模板类则相当于按照模板描绘,其形状相同,只是颜色各异。

**软件工程知识 12.1** 模板是C++ 软件可重用的重要功能之一。

本章将介绍一些函数模板和类模板,并考察模板与其他C++ 特性(如重载、继承、友元和静态成员)之间的关系。

这里介绍的模板机制的设计和细节以 Bjarne Stroustrup 的论文《Parameterized Types for C++》为基础,这篇论文于 1988 年 10 月发表于科罗拉多州丹佛市举办的 USENIX C++ 会议。

模板问题本身既丰富又复杂,本章只作简要介绍。第 20 章(即本书篇幅最长的一章)将深入介绍模板容器类、迭代器和标准模板库(STL)算法。第 20 章有几十个基于模板的“活代码”,演示了更复杂的模板编程技术。

## 12.2 函数模板

重载函数通常用于对不同的数据类型执行相似的操作。如果对每种数据类型执行相同的操作,使用函数模板来完成将更为简便。程序员只编写一次函数模板的定义。根据调用

函数时提供的参数类型,编译器会产生相应的目标代码函数以正确处理每种类型的调用。C语言中,这是用预处理程序指令#define 创建的宏来完成的(参见第17章)。但是,宏可能有负面影响,并使编译器无法执行类型检查。函数模板和宏一样简洁,但能让编译器执行全面的类型检查。

**测试和调试提示 12.1** 函数模板和宏一样,支持软件的重用。但与宏不同的是,函数模板还可以消除许多类型错误,因为C++提供了安全而全面的类型检查。

所有的函数模板定义都以关键字 template 开头,然后是用尖括号(< >)括起来的形式类型参数表。每一个形式类型参数之前都有关键字 class 或 typename,例如

```
template<class T>
```

或

```
template<typename ElementType>
```

或

```
template<class BorderType, class FillType>
```

模板定义的形式类型参数表(参数可以是内部类型和自定义类型)用于指定传递给函数的参数类型、函数返回类型和声明函数中变量。该函数的方式定义与其他函数的定义类似。注意,用于指定函数模板类型参数的关键字 class 实际上表示“所有内部类型或用户自定义类型”。

**常见编程错误 12.1** 函数模板的每个形式类型参数前不放置关键字 class(或新的关键字 typename)。

接下来看图 12.1 中的 printArray 函数模板,该函数模板的用法参见图 12.2 中的完整程序。

```
1 template< class T >
2 void printArray( const T *array, const int count )
3 {
4     for ( int i = 0; i < count; i++ )
5         out << array[ i ] << " ";
6
7     cout << endl;
8 }
```

图 12.1 函数模板 printArray

该函数模板把惟一的形式类型参数 T(T 可以是任意有效的标记符)声明为函数 printArray 打印的数组类型;T \* 就是类型参数。编译器检测到程序源代码中调用了函数 printArray 时,用 printArray 的第一个参数的类型代替整个模板定义中的 T,并创建一个完整的用于打印指定类型数组的模板函数,并编译这个新建的函数。图 12.2 中,实例化了 3 个 printArray 函数,它们分别需要一个 int 类型的数组、一个 double 类型的数组和一个 char 类型的数组。int 类型数组的实例化过程如下所示

```
void printArray(const int *array, const int count)
{
    for ( int i = 0; i < count; i++ )
```

```

        cout << array[i] << " ";
    count << endl;
}

```

模板函数中的每一个形式类型参数至少应在函数参数表中出现一次。形式类型参数的名字可以只在模板函数的形式参数表中出现一次。模板函数中的形式类型参数名可以不惟一。

```

1  //Fig 12.2: fig12_02.cpp
2  //Using template functions
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  template< class T >
9  void printArray( const T *array, const int count )
10 {
11     for ( int i = 0; i < count; i++ )
12         cout << array[ i ] << " ";
13
14     cout << endl;
15 }
16
17 int main()
18 {
19     const int aCount = 5, bCount = 7, cCount = 6;
20     int a[ aCount ] = { 1, 2, 3, 4, 5 };
21     double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
22     char c[ cCount ] = "HELLO"; //6th position for null
23
24     cout << "Array a contains:" << endl;
25     printArray( a, aCount ); //integer template function
26
27     cout << "Array b contains:" << endl;
28     printArray( b, bCount ); //double template function
29
30     cout << "Array c contains:" << endl;
31     printArray( c, cCount ); //character template function
32
33     return 0;
34 }

```

输出结果:

```

Array a contains:
1 2 3 4 5
Array b contains:
1.2 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O

```

图 12.2 模板函数用法示例

图 12.2 中的程序演示了模板函数 `printArray` 的用法。程序首先实例化 `int` 数组 `a`、`double` 数组 `b` 和 `char` 数组 `c`，长度分别为 5, 7, 6。然后调用 `printArray` 打印各个数组，一次用 `a` 的第一个参数，类型为 `int *`；一次用 `b` 的第一个参数，类型为 `double *`；一次用 `c` 的第一个参数，类型为 `char *`。例如，语句

```
printArray( a, aCount );
```

令编译器实例化 `printArray` 模板函数，类型参数 `T` 为 `int`。语句

```
printArray( b, bCount );
```

令编译器实例化第二个 `printArray` 模板函数，类型参数 `T` 为 `double`。语句

```
printArray( c, cCount );
```

令编译器实例化第 3 个 `printArray` 模板函数，类型参数 `T` 为 `char`。

本例中，由于采用了模板，程序员就不必用原型

```
void printArray( const int *, const int );
void printArray( const double *, const int );
void printArray( const char *, const int );
```

编写 3 个重载函数。除了类型 `T` 外，代码都相同。

**性能提示 12.1** 模板凸显了软件重用的好处。但记住，尽管模板只编写一次，但程序中仍须实例化多个模板函数和模板类的副本。这些副本会占用大量内存。

## 12.3 重载模板函数

模板函数与重载密切相关。通过函数模板产生的相关函数均同名的，因此编译器会用重载的方法调用相应函数。

函数模板本身可以用多种方式重载。我们可以提供其他函数模板，指定不同参数的相同函数名。例如，图 12.2 中的 `printArray` 函数模板可以用另一个 `printArray` 函数模板重载，用参数 `lowSubscript` 和 `highSubscript` 指定要打印的数组部分（参见练习题 12.4）。

函数模板也可以用其他非模板函数（名称相同而参数不同）重载。例如，图 12.2 的 `printArray` 函数模板可以用一个非模板函数重载，指定以整齐的表格式分栏打印字符串数组（参见练习题 12.5）。

**常见编程错误 12.2** 如果使用用户自定义类的类型调用模板，而模板对该类型对象使用 `==`，`+`，`<=` 等操作符，那么这些操作符就需要重载。如果不重载这些操作符，就会发生错误，因为编译器在这些函数不存在的情况下，仍然会调用这些重载的操作符函数。

调用函数时，编译器通过匹配过程确定调用的函数。首先，编译器寻找和使用最匹配函数名和参数类型的函数调用。如果找不到，编译器将检查是否可以用函数模板产生匹配函数名和参数类型的模板函数。找到匹配的函数模板，编译器就生成和使用相应的模板函数。

**常见编程错误 12.3** 编译器通过匹配过程确定调用的函数，找不到匹配或找到多个匹配，就会出现编译错误。

## 12.4 类模板

堆栈(即一种数据结构,其中的数据项以某种顺序插入但以后进先出的顺序提取)与栈中数据项的类型无关,这一点不难理解。但是,用程序实现堆栈时还必须提供数据类型,这为实现软件的重用性创造了绝佳的机会。所需的方法是描述一个普遍意义上的堆栈,然后建立这个类的实例类。所建的实例类虽然是通用类的副本,但是它具有指定的类型。C++的模板类使提供了这种功能。

**软件工程知识 12.2** 类模板通过实例化通用类的特定版本增强了软件的重用性。

模板类通常也称为参数化类型,这是因为模板类需要一种或多种类型参数来说明如何定制通用类的模板以形成指定的模板类。

程序员要想生成多种模板类,只需编写一个通用类模板的定义。在需要用模板建立一个新类时,也只需一种简洁的表达方式,让编译器写出程序需要的模板类的源代码。例如,堆栈类的模板可以作为编写各种类型堆栈的基础(如 float 类型、int 类型或 char 类型的堆栈等)。

注意图 12.3 程序中 Stack(堆栈)的类模板定义。模板类与传统的类定义没有多大区别,只不过模板类的定义均以下面的首部(第 6 行)

```
template < class T >
```

开始,这行代码表明这是类模板定义的“标志”,它带有一个类型参数 T(表示所要建立的 Stack 类的类型)。程序员不一定要用标识符 T,可选用其他任何标识符。Stack 中存储的元素类型在 Stack 类首部和成员函数定义中统一表示为 T。稍后将介绍如何使 T 与特定类型(如 double 或 int)相关联。随同该 Stack 一起使用的非原始数据类型有两类容器,它们必须有一个默认的构造函数,而且必须支持赋值操作符。如果随该 Stack 一起使用的类对象包含动态分配内存,就应该为这类对象重载赋值操作符,详情参见第 8 章。

```
1 //Fig.12.3: tstack1.h
2 //Class template Stack
3 #ifndef TSTACK1_H
4 #define TSTACK1_H
5
6 template < class T >
7 class Stack {
8 public:
9     Stack( int = 10 );    //default constructor (stack size 10)
10    ~Stack() { delete [] stackPtr; } //destructor
11    bool push( const T& ); //push an element onto the stack
12    bool pop( T& );        //pop an element off the stack
13 private:
14     int size;             //# of elements in the stack
15     int top;              //location of the top element
16     T *stackPtr;          //pointer to the stack
17
18     bool isEmpty() const { return top == -1; } //utility
```

```

19     bool isFull() const { return top == size - 1; } //functions
20 };
21
22 //Constructor with default size 10
23 template< class T >
24 Stack< T >::Stack( int s )
25 {
26     size = s > 0 ? s : 10;
27     top = -1;           //Stack is initially empty
28     stackPtr = new T[ size ]; //allocate space for elements
29 }
30
31 //Push an element onto the stack
32 //return 1 if successful, 0 otherwise
33 template< class T >
34 bool Stack< T >::push( const T &pushValue )
35 {
36     if ( ! isFull() ) {
37         stackPtr[ ++top ] = pushValue; //place item in Stack
38         return true; //push successful
39     }
40     return false; //push unsuccessful
41 }
42
43 //Pop an element off the stack
44 template< class T >
45 bool Stack< T >::pop( T &popValue )
46 {
47     if ( ! isEmpty() ) {
48         popValue = stackPtr[ top - - ]; //remove item from Stack
49         return true; //pop successful
50     }
51     return false; //pop unsuccessful
52 }
53
54 #endif

```

图 12.3 类模板 Stack——tstack1. h

```

55 //Fig. 12.3: fig12_03.cpp
56 //Test driver for Stack template
57 #include <iostream>
58
59 using std::cout;
60 using std::cin;
61 using std::endl;
62
63 #include "tstack1.h"
64
65 int main()
66 {

```

```
67     Stack< double > doubleStack( 5 );
68     double f = 1.1;
69     cout << "Pushing elements onto doubleStack\n";
70
71     while ( doubleStack.push( f ) ) { //success true returned
72         cout << f << ' ';
73         f += 1.1;
74     }
75
76     cout << "\nStack is full. Cannot push " << f
77         << "\n\nPopping elements from doubleStack\n";
78
79     while ( doubleStack.pop( f ) ) //success true returned
80         cout << f << ' ';
81
82     cout << "\nStack is empty. Cannot pop\n";
83
84     Stack< int > intStack;
85     int i = 1;
86     cout << "\nPushing elements onto intStack\n";
87
88     while ( intStack.push( i ) ) { //success true returned
89         cout << i << ' ';
90         ++i;
91     }
92
93     cout << "\nStack is full. Cannot push " << i
94         << "\n\nPopping elements from intStack\n";
95
96     while ( intStack.pop( i ) ) //success true returned
97         cout << i << ' ';
98
99     cout << "\nStack is empty. Cannot pop\n";
100     return 0;
101 }
```

#### 输出结果:

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. cannot push 6.6
```

```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop
```

```
Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11
```

```
Popping elements from intStack
```

```
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

图 12.3 类模板 Stack——fig12\_03.cpp

接下来分析一段驱动程序(函数 main),该程序是对 Stack 类模板的一个练习(参见图 12.3 的输出结果)。程序开始使实例化一个长度为 5 的对象 doubleStack。该对象声明为 Stack <double> 类的对象。编译器自动把模板中的参数类型 T 替换为 double 以生成 double 类型的 Stack 类的源代码。尽管程序看不到这段源代码,但仍会将其放入源代码编译。

然后程序将 1.1, 2.2, 3.3, 4.4 和 5.5 这几个 double 值成功压入堆栈 doubleStack。试图将第 6 个值压入堆栈时, push 循环中止(栈已经满了,因为它只能容纳 5 个元素)。

程序再以后进先出(LIFO)的顺序将这 5 个元素弹出堆栈。在试图弹出第 6 个元素时,出栈循环中止,因为时栈已经空了。

接下来,程序用声明语句

```
stack<int> instack;
```

实例化一个 int 类型的堆栈 intStack。由于没有指定堆栈的长度,所以取默认构造函数(第 24 行)中的默认值 10 作为堆栈的长度。重复上述操作,用循环结构不断向 intStack 中压入整数值,直到栈满为止,然后再循环从堆栈中弹出数值,直到栈空为止。

在类模板首部以外的成员函数定义均以代码

```
template <class T>
```

开头,成员函数的定义与普通成员函数的定义相似,只是 Stack 元素的类型要用类型参数 T 表示。二元作用域分辨符和 Stack <T> 类模板将成员函数的定义与正确的类模板范围对应起来。本例中,类名是 Stack <T>。当 doubleStack 被实例化为 Stack <double> 类时,Stack 的构造函数用 new 建立了一个表示堆栈的 double 类型数组(第 28 行)。因此,对于语句

```
stackPtr = new T[size];
```

编译器将在模板类 Stack <double> 中生成代码

```
StackPtr = new double[Size];
```

注意图 12.3 中函数 main 的代码(即 main 上半部分的 doubleStack 操作和 main 下半部分的 intStack 操作)基本相同。这里也可以用函数模板。图 12.4 完成的程序用函数模板 testStack 完成图 12.3 的工作,将一系列值压入 Stack <T> 中并从 Stack <T> 中弹出数值。函数模板 testStack 用参数 T 表示 Stack <T> 中保存的数据类型。该函数模板取 4 个参数: Stack <T> 类型对象的引用、类型为 T 的值用作压入 Stack <T> 的第一个值、类型为 T 的值用作压入 Stack <T> 的增量值以及 const char \* 类的字符串表示输出的 Stack <T> 对象名。函数 main 只是实例化 Stack <double> 类型对象 doubleStack 和实例化 Stack <int> 类型对象 intStack,如下所示(第 42 行和第 43 行)

```
testStack(doubleStack, 1.1, 1.1, "doubleStack");
testStack(intStack, 1, 1, "intStack");
```

注意,图 12.4 中的输出结果与图 12.3 中的输出结果完全一样。

```
1 //Fig. 12.4: fig12_04.cpp
2 //Test driver for Stack template.
3 //Function main uses a function template to manipulate
4 //objects of type Stack < T >.
```



```
5 #include <iostream>
6
7 using std::cout;
8 using std::cin;
9 using std::endl;
10
11 #include "tstack1.h"
12
13 //Function template to manipulate Stack < T >
14 template< class T >
15 void testStack(
16     Stack< T > &theStack,    //reference to the Stack < T >
17     T value,                //initial value to be pushed
18     T increment,            //increment for subsequent values
19     const char *stackName ) //name of the Stack < T > object
20 {
21     cout << "\nPushing elements onto " << stackName << '\n';
22
23     while ( theStack.push( value ) ) { //success true returned
24         cout << value << ' ';
25         value += increment;
26     }
27
28     cout << "\nStack is full. Cannot push " << value
29         << "\n\nPopping elements from " << stackName << '\n';
30
31     while ( theStack.pop( value ) ) //success true returned
32         cout << value << ' ';
33
34     cout << "\nStack is empty. Cannot pop\n";
35 }
36
37 int main()
38 {
39     Stack< double > doubleStack( 5 );
40     Stack< int > intStack;
41
42     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
43     testStack( intStack, 1, 1, "intStack" );
44
45     return 0;
46 }
```

输出结果:

Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5

Stack is full. Cannot push 6.6

Popping elements from doubleStack

5.5 4.4 3.3 2.2 1.1

```

Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

```

图 12.4 把模板对象 Stack 传给函数模板

## 12.5 类模板与无类型参数

前一节的 Stack 类模板只采用了模板首部的类型参数。它还可以使用无类型参数 (non-type parameter), 可以有默认参数, 一般作为常量处理。例如, 模板首部可以取 int elements 参数, 如下所示

```
template< class T,int elements > //note non-type parameter
```

然后声明

```
Stack<double,100 > mostRecentSalesFigures;
```

在编译时实例化 100 个元素的 Stack 模板类 mostRecentSalesFigures (使用 double 值)。这个模板类的类型为 Stack<double,100>。类的首部可以包含 private 数据成员, 数组声明如下

```
T StackHolder[elements]; //array to hold stack contents
```

**性能提示 12.2** 如果可能, 应在编译时指定容器类 (如数组类和堆栈类) 的长度 (可能通过非类型模板长度参数) 以消除 new 动态生成空间所引起的执行时开销。

**软件工程知识 12.3** 如果可能, 应在编译时指定容器类的长度 (可能通过非类型模板长度参数) 以避免 new 无法取得所需内存时引发致命的运行时错误。

练习题要求用无类型参数为第 8 章开发的 Array 类创建模板。这个模板将用编译时指定类型的指定元素个数实例化 Array 对象, 而不是运行时动态生成 Array 对象的空间。

如果某类特殊类型的类不能与普通类模板匹配, 则可以为该特殊类型的类重新定义类模板。例如, 可以用一个 Array 类模板实例化任何类型的数组。程序员可以控制某个类型 Array 类的实例化, 如 Martian, 只要建立类名为 Array<Martian> 的新类即可。

## 12.6 模板与继承

模板与继承具有以下关系:

- 类模板可以从模板类派生;
- 类模板可以从非模板类派生;
- 模板类可以从类模板派生;
- 非模板类可以从类模板派生。

## 12.7 模板与友元

不难发现,函数和整个类都可以声明为非模板类的友元。使用类模板,可以声明各种各样的友元关系。友元可以在类模板与全局函数间、另一个类(可能是模板类)的成员函数间或整个类中(可能是模板类)建立。建立这种友元关系所需的符号不太容易掌握。

已用语句

```
template <class T> class X
```

声明的 X 类的类模板中,友元关系声明

```
friend void f1();
```

使函数 `f1` 成为从上述类模板实例化的每个模板类的友元。

用语句

```
template <class T> class X;
```

声明的 X 类的类模板内,对类似 `float` 的特定类型 T 而言,友元声明

```
friend void f2(x <T> &);
```

只能使函数 `f2(X <float> &)` 成为 `X <float>` 的友元。

在类模板中,可以把另一个类的成员函数声明为类模板生成的任何模板类的友元。只须用类名和二元作用域分辨符指定其他类的成员函数名。例如,语句

```
template <class T> class X;
```

声明的 X 类的类模板内,友元关系声明:

```
friend void A::f4();
```

使 A 类的成员函数 `f4` 成为上述类模板实例化的任何模板类的友元。

用语句:

```
template <class T> class X;
```

声明的 X 类的类模板内对类似于 `float` 的特定类型 T 而言,友元声明:

```
friend void C <T>::f5();
```

将令成员函数

```
C <float>::f5(X <float> &);
```

成为 `X <float>` 模板类的友元函数。

X 类的类模板声明

```
template <class T> class X;
```

可以声明第二个 Y 类,如下所示

```
friend class Y;
```

将使 Y 类的每个成员函数成为 X 的类模板产生的每个模板类的友元。

X 类的类模板声明

```
template <class T> class X;
```

可以声明第二个 Z 类,如下所示

```
friend class Z <T>;
```

然后在使模板类用特定类型 T(如 `float`) 实例化时, `class Z <float>` 的所有成员都将成为模板类 `X <float>` 的友元。

## 12.8 模板与静态数据成员

非模板类中,类的所有对象共享一个静态(static)数据成员,静态数据成员应在文件范围内初始化。

通过类模板实例化的每个模板类都有自己的类模板静态数据成员,该模板类的所有对象共享一个静态数据成员。和非模板类的静态数据成员一样,模板类的静态数据成员也应在文件范围内初始化。每个模板类都有自己的类模板的静态数据成员副本。

## 12.9 小结

- 模板使我们可用单个代码段指定一组相关函数(称为模板函数)或一组相关类(称为模板类)。
- 程序员只编写一次函数模板定义。基于调用函数时提供的参数类型,C++自动产生单独的函数正确处理每种类型的调用。这些都是利用程序源代码的剩余空间进行编译。
- 所有函数模板定义都以关键字 `template` 开头,后接用尖括号(`< >`)括起来的形式参数表。函数模板的每个形式类型参数之前应有关键字 `class` (或新的关键字 `typename`)。关键字 `class` 指定函数模板的类型参数,实际上表示“任何内部类型或用户自定义类型”。
- 模板定义的形式参数可用于指定传递给函数的参数类型、函数返回类型和声明函数中的变量。
- 模板的形式参数表中形式参数的名称可以只出现一次。同一个形式参数名可用于多个模板函数。
- 函数模板本身可以用多种方式重载。我们可以提供其他函数模板,指定参数不同的相同函数名。函数模板也可以用其他非模板函数(同名而不同参数)重载。
- 类模板提供了描述类和实例化类(即该通用类指定类型的版本)的方法。
- 为了说明如何定制通用类模板以形成指定的模板类,类模板需要类型参数,所以类模板也常常称为参数化类型。
- 要使用模板类的程序员只需编写一个类模板。在需要用模板建立一个新的指定类型的类时,程序员只需要用一种简洁的表示方法,编译器就会写出该模板类的源代码。
- 类模板的定义与普通的类定义没什么不同,除了使用 `template < class T >` 指明这是一个带类型参数 `T`(指明创建的类的类型)的类模板定义。在类首部和成员函数的定义中,类型 `T` 是一个通用的类型名。
- 在类模板首部以外的成员函数定义都要以 `template < class T >` 开头。接着,成员函数的定义与普通成员函数的定义相似,只是类中的数据通常用类型参数 `T` 表示。二元作用域分辨符总是把成员函数的定义与正确的类范围对应起来。

- 类模板首部也可以使用无类型参数。
- 特定类型的类可以重定义该类型的类模板。
- 类模板可以从模板类派生。类模板可以从非模板类派生。模板类可以从类模板派生。非模板类可以从类模板中派生。
- 函数和整个类都可以声明为非模板类的友元。使用类模板,可以声明各种各样的友元关系。友元可以在类模板与全局函数间、另一个类(可能是模板类)的成员函数间或整个类中(可能是模板类)建立。
- 从类模板实例化的每个模板类有自己的类模板的静态数据成员,该模板类的所有对象共享一个静态数据成员。和非模板类的静态数据成员一样,模板类的静态数据成员也应在文件范围内初始化。
- 每个模板类有该类模板的静态数据成员副本。

## 本章术语

|                                            |                                            |
|--------------------------------------------|--------------------------------------------|
| angle brackets 尖括号                         | static data member of a class template     |
| class template name 类模板名                   | 类模板的静态数据成员                                 |
| class template 类模板                         | static data member of a template class     |
| formal parameter in a template header      | 模板类的静态数据成员                                 |
| 模板首部中的形式参数                                 | static member function of a class template |
| friend of a template 模板的友元                 | 类模板的静态成员函数                                 |
| function template declaration 函数模板的声明      | static member function of a template class |
| function template definition 函数模板的定义       | 模板类的静态成员函数                                 |
| function template 函数模板                     | template argument 模板实参                     |
| keyword class in a template type parameter | template class member function 模板类成员函数     |
| 模板类型参数中的关键字 class                          | template class 模板类                         |
| keyword template 关键字 template              | template function 模板函数                     |
| non-type parameter in a template header    | template name 模板名                          |
| 模板首部中的无类型参数                                | template parameter 模板形参                    |
| overloading a template function 重载模板函数     | type parameter in a template header        |
| parameterized type 参数化类型                   | 模板首部的类型参数                                  |

## 常见编程错误

- 12.1 函数模板的各形式类型参数前不添加关键字 class(或新的关键字 typename)。
- 12.2 如果使用用户自定义类的类型调用模板,而模板对该类型对象使用 ==、+、<= 等操作符,这些操作符就需要重载。如果不重载这些操作符,就会发生错误,因为编译器在这些函数不存在的情况下仍然会调用这些重载的操作符函数。
- 12.3 编译器通过匹配过程确定调用的函数,找不到匹配或找到多个匹配,就会出现编译错误。

## 性能提示

- 12.1 模板凸显了软件重用的好处。但请记住,尽管模板只编写一次,但程序中仍须实例化多个模板函数和模板类的副本。这些副本会占用大量内存。

- 12.2 如果可能,应在编译时指定容器类(如数组类和堆栈类)的长度(可能通过非类型模板长度参数)以消除 new 动态生成空间所引起的执行时开销。

### 软件工程知识

- 12.1 模板是C++ 软件重用性的重要功能之一。  
 12.2 类模板通过实例化采用类的特定版本增强了软件的重用性。  
 12.3 如果可能,应在编译时指定容器类的长度(可能通过非类型模板长度参数)以避免 new 无法取得所需内存时引发致命的运行时错误。

### 测试和调试提示

- 12.1 函数模板和宏一样支持软件的重用。但与宏不同的是,函数模板还可以消除许多类型错误,因为C++ 提供了安全而全面的类型检查。

### 自测题

- 12.1 判断正误。如果不正确,请说明原因。
- 函数模板的友元函数必须是模板函数。
  - 如果通过带单个静态数据成员类模板产生几个模板类,则每个模板类共享类模板静态数据成员的一个副本。
  - 模板函数可以用同名的另一个模板函数重载。
  - 形式参数的名字可以只在模板函数的形式参数表中出现一次。同一个形式参数名只能用于一个模板函数。
  - 关键字 class 指定函数模板类型参数,实际上表示“任何用户自定义类型”。
- 12.2 填空题:
- 模板使我们可用单独的个代码段指定一组相关函数(称为\_\_\_\_\_)或一组相关类(称\_\_\_\_\_ )。
  - 所有的函数模板定义都是以关键字\_\_\_\_\_ 开始的,该关键字之后是用\_\_\_\_\_ 括起来的形式参数表。
  - 从一个函数模板产生的相关函数都同名,因此编译器用\_\_\_\_\_ 的解决方法调用相应函数。
  - 类模板也称为\_\_\_\_\_ 类型。
  - \_\_\_\_\_ 分辨符和模板类名一起将每个成员函数定义与类模板的范围相关联。
  - 和非模板类的静态数据成员一样,模板类的 static 数据成员也应在\_\_\_\_\_ 范围内初始化。

### 自测题答案

- 12.1 a) 错误。也可以用非模板函数。  
 b) 错误。每个模板类有自己的静态数据成员副本。  
 c) 正确。  
 d) 错误。模板函数间的形式参数名无须惟一。  
 e) 错误。这里的关键字 class 也允许内部类型的参数类型。

- 12.2 a) 模板函数、模板类。  
 b) template、尖括号(< >)。  
 c) 重载。  
 d) 参数化。  
 e) 二元作用域。  
 f) 文件。

### 练习题

- 12.3 以图 5.15 的排序程序为基础,编写函数模板 bubbleSort。编写一个驱动程序,输入、排序及输出 int 数组和 float 数组。
- 12.4 重载图 12.2 的函数模板 printArray,使其取另外两个整数参数 int lowSubscript 和 int highSubscript。调用这个函数只打印数组中的指定部分。验证 lowSubscript 和 highSubscript。如果其中一个值超界或 highSubscript 小于或等于 lowsubscript,重载的 printArray 函数就会返回 0,否则 printArray 返回打印的元素个数。然后修改 main,对数组 a,b,c 使用两个版本的 printArray。一定要测试 printArray 两个版本的各种可能情况。
- 12.5 用非模板版本重载图 12.2 的函数模板 printArray,使其以整齐的表格式分栏格式打印字符串数组。
- 12.6 编写判断函数 isEqualTo 的简单函数模板,用相等操作符比较它的两个参数,相等就返回 1,不相等则返回 0。使用这个函数模板,令程序中各种内部类型调用 isEqualTo。现在编写程序的另一种形式,对用户自定义类的类型调用 isEqualto,但不重载相等操作符。运行这个程序时会发生什么情况?重载相等操作符(用操作符函数 operator ==),运行这个程序时会发生什么情况?
- 12.7 用无类型参数 numberOfElements 和类型参数 elementType 生成第 8 章开发的 Array 类模板。这个模板按编译时指定个数的指定元素类型实例化 Array 对象。
- 12.8 编写使用类模板 Array 的程序,模板可以实例化任何元素类型的 Array 对象。用 float 元素的 Array(class Array<float>)重定义模板。驱动程序演示通过模板实例化 int 类型的 Array,并使用 class Array<float> 中提供的定义实例化 float 类型的 Array。
- 12.9 说明模板函数与函数模板的不同。
- 12.10 类模板与模板类哪个像是能够绘制形状 of 模板?为什么?
- 12.11 函数模板与重载有什么关系?
- 12.12 为什么要用函数模板代替宏?
- 12.13 使用函数模板与类模板可能造成哪些性能问题?
- 12.14 编译器通过匹配过程确定函数调用时调用哪个模板函数。什么情况下进行匹配会造成编译错误?
- 12.15 为什么类模板也称为参数化类型?
- 12.16 说说为什么要在 C++ 程序中使用下列语句。
- ```
Array<Employee> workerList(100);
```
- 12.17 分析练习题 12.16 的答案,说说为什么要在 C++ 程序中使用下列语句。
- ```
Array<Employee> workerList;
```
- 12.18 说说为什么要在 C++ 程序中使用下列语句。

```
template< class T > Array< T >::Array( int s )
```

12.19 为什么数组、堆栈之类的容器类模板通常使用无类型参数?

12.20 说明如何提供特定类型的类重定义该类型的类模板。

12.21 说明类模板与继承的关系。

12.22 假设类模板的首部如下:

```
template< class T1 > class C1
```

请说明类模板首部中用下列友元声明时的友元关系。以 f 开头的标识符是函数,以 C 开头的标识符是类,以 T 开头的标识符是任何类型(即内部类型或类类型)。

a) friend void f1();

b) friend void f2 (C1<T1>&);

c) friend void C2::f4();

d) friend void C3<T1>::f5(C1<T1>&);

e) friend class C5;

f) friend class C6<T1>;

12.23 假设类模板 Employee 有静态数据成员 count。如果从类模板实例化 3 个模板类,会产生多少个静态数据成员的副本? 各个副本的使用有什么限制(如果有的话)?



# 第 13 章 异常处理

## 学习目标

- 能用 try, throw 和 catch 分别监视、指定和处理异常
- 能处理未捕捉和未预料到的异常
- 能处理 new 失败
- 能利用 auto\_ptr 防止内存泄漏
- 能理解标准异常层次结构

## 13.1 简介

本章介绍异常处理。C++ 具有强大的扩展能力,但也大大增加错误产生的可能性和种类。本章介绍的特性有助于程序员编写出更清晰、更健全、更具容错性的程序。目前利用这类技术开发的新系统已产生了积极效果。另外还将介绍何时不宜使用异常处理。

本章介绍的异常处理样式和细节以 Andrew Koenig 和 Bjarne Stroustrup 的论文《Exception Handling for C++ (修订版)》中的内容为基础;该论文于 1990 年 4 月发表于美国旧金山市举行的 USENIX C++ 会议。

错误处理代码的性质与数量一般不尽相同。具体说来与应用的软件系统和该软件是否已经发布有关。通常情况下,商业化产品提供的错误处理代码要比自用软件提供的更多。

处理错误的方法也多种多样。通常,错误处理代码分布在整个系统代码中。任何代码可能出错的地方都进行错误处理。这样可使程序员在阅读代码时直接看到错误处理情况,确定是否实现了正确的错误检查。

但是,这种机制有“污染”正常程序代码之嫌,使应用程序代码变得更晦涩,难以看出代码功能是否得以正确实现,如此一来,更加大了理解和维护代码的难度。

常见的异常例子有:new 无法取得所需内存、数组下标超界、运算溢出、除数为 0 和无效函数参数。

C++ 特有的异常处理使程序员可以抛弃程序执行“主线”中的错误处理代码,增强程序的可读性和可维护性。C++ 式的异常处理可以捕捉所有类型的异常、捕捉特定类型的所有异常和捕捉相关类型的所有异常。这样便大大减少了程序不能捕捉错误的可能性,使程序更健壮。异常处理使程序可以捕捉和处理错误,而不是任其发生最后导致恶果。程序员若不提供处理致命错误的机制,程序会被终止。

异常处理被设计用来处理同步错误如除数为 0(在程序执行除法指令时发生)。在程序执行除法之前,异常处理首先检查除数,如果除数为 0,则抛出异常信息。

异常处理并不负责处理异步情况,如磁盘 I/O 完成、网络消息到达、鼠标单击等等,这些

情况最好用其他方法处理,如中断处理。

异常处理用于从导致异常的错误中恢复系统。这一恢复过程即异常处理过程。

异常处理通常用于发现错误的部分与处理错误的部分不在同一位置(不同范围)时。与用户进行交互式对话的程序不能用异常来处理输入错误。

异常处理特别适合用于程序无法恢复但又需要提供有序清理,以使程序可以正常结束时。

**良好编程习惯 13.1** 发生范围与处理范围不同的错误可以使用异常处理。发生范围与处理范围相同的错误,则应该用其他方法处理。

**良好编程习惯 13.2** 使用异常处理时,尽量避免错误处理以外的其他用途的处理,这样可使程序更清晰。

传统程序控制避免使用异常处理还有别的原因。异常处理用于错误处理,不是用于处理程序准备终止时常有的活动。如果将两者混合,C++ 编译器的编写人员不能对待常用应用程序代码一样,实现最优化性能。

**性能提示 13.1** 尽管异常处理可以进行错误处理以外的工作,但将两种用途混用会降低性能。

**性能提示 13.2** 编译器实现异常处理时,通常会在异常不发生时将异常处理代码开销降到极小或为0,发生异常时,则执行关的系统开销。当然,异常处理代码无疑会使程序占用更多内存。

**软件工程知识 13.1** 传统控制结构的控制流通常比使用了异常的控制流更清晰、更有效。

**常见编程错误 13.1** 传统程序控制使用异常处理的另一危险是堆栈解退,异常发生之前分配的资源可能无法释放。这个问题可以通过认真编程来避免。

异常处理可以提高程序的容错性。由于编写错误处理代码较为轻松,因此程序员更愿意提供错误处理代码。异常处理还提供了各种不同方法捕捉异常,如根据类型捕捉异常,甚至指定捕捉任何类型的异常。

目前大多数程序都只支持单线程执行。WindowsNT, OS/2 和各种 UNIX 操作系统越来越重视多线程。本章讨论的技术同样适用于多线程程序,但我们不打算专门介绍多线程程序。

我们将介绍如何处理未捕捉异常,考虑未捕捉异常的处理方式,介绍如何用公共异常基类派生的异常类表示相关异常。

随着C++ 标准的发展,C++ 的异常处理特性也得以广泛应用。在几十、上百人参与的大型软件项目中,标准化尤为重要,每个人参与系统的不同组件,这些组件要在整个系统中相互影响以实现既定的功能。

**软件工程知识 13.2** 异常处理适用于分组件开发的系统。异常处理使组件组合更为容易。每个组件可以独立于其他异常处理范围之外,执行自己的异常检测。

异常处理可以认为是从函数返回控制或从代码段返回的另一种方法。通常,有异常发

生时,处理该异常的必然是产生异常的函数调用者、函数调用者的调用者或更深层的调用者的之一。

## 13.2 何时使用异常处理

虽然无法阻止程序员用异常来代替程序控制,但异常处理应当只用于异常情况,用于处理程序组件中与其他异常处理没有直接关系的异常,用于处理在函数、库、类等常用软件组件中广泛使用的而组件本身不处理的异常情况,及用于在大型系统中以统一方式处理异常。

**良好编程习惯 13.3** 对程序本身容易处理的简单而又局部性错误,应用传统错误处理方法而避免使用异常处理。

**软件工程知识 13.3** 对于库操作,库函数调用者通常用特定错误处理方法处理库函数中产生的异常。库函数很难进行可满足用户独特需求的错误处理。所以,我们说异常适合处理库函数产生的错误。

## 13.3 其他错误处理方法

本章之前我们已经介绍了许多错误处理方法,一些处理方法总结如下:

- 用 `assert` 测试编码和设计错误。如果返回 `false`,则程序终止,改正代码。这样的方法非常适用于调试;
- 对公开发布的软件产品和执行关键任务的专用软件而言,忽略异常绝不是明智之举。自用软件则不然,通常允许存在许多错误;
- 退出程序,这当然可以防止程序继续运行或产生错误结果。实际上,对于许多错误类型,特别是对于能让程序运行完毕的非致命错误,这是个好办法,因为让程序运行完毕很可能使程序员误以为程序运行正常。但这种方法不适合执行关键任务的所有应用程序。资源问题也不容忽视,程序一旦取得资源,也应该在正常释放资源之后终止;

**常见编程错误 13.2** 退出程序会使其他程序无法使用其资源,从而造成资源泄漏。

- 设置一些错误说明符。问题是程序不一定在发生错误的所有地方都检查这些错误说明符;
- 测试错误条件、发出错误消息和调用 `exit`,向程序环境传递相应的错误代码;
- `setjump` 和 `longjump`。这些 `<csjtmp.h>` 提供的库函数,可以指定从深层嵌套函数立即转入错误处理器。如果没有 `setjump/longjump`,程序要执行几层返回才能从深层嵌套函数退出。这种方法可以转入某个错误处理程序。但这种方法在C++中有潜在危险,因为它解退堆栈时不调用自动对象的析构函数,可能会造成严重问题;
- 某些特定错误有专门的处理功能。例如,`new` 无法分配内存时,可以用 `new _ handler` 函数处理错误。通过提供函数名作为 `set _ new _ handler` 的参数可以改变这个函数。

## 13.4 C++ 异常处理基础: try, throw 和 catch

C++ 异常处理用于错误检测函数无法处理错误的情况。错误检测函数抛出异常 (throw an exception), 但不能保证有相关的异常处理程序 (每一个异常处理函数专门用于处理某一类的异常)。如果有, 异常处理程序将捕捉并处理这个异常。如果没有该类异常相关的异常处理程序, 程序则会终止。

程序员在 try 块中放入出错时产生异常的代码。try 块后面是一个或几个 catch 块。每个 catch 块指定捕捉和处理一种异常, 而且每个 catch 块包含一个异常处理程序。如果异常与 catch 块中的参数类型相符, 则执行该 catch 块的代码。找不到相应异常处理程序, 则调用 terminate 函数 (默认调用函数 abort)。

抛出异常时, 程序控制离开 try 块, 遍历 catch 块以搜索相应异常处理程序 (稍后将介绍如何形成相应的异常处理程序。) 如果 try 块中没有抛出异常, 则跳过该块的异常处理程序, 直到最后一个 catch 块遍历完成, 程序继续执行。

我们可以指定一个函数抛出的异常, 也可以指定函数抛出任何异常。

异常可以在函数的 try 块中抛出, 也可以从 try 块直接或间接调用的函数抛出异常。执行抛出的点称为抛出点 (throwpoint)。抛出点也用于描述抛出表达式本身。一旦有异常抛出, 程序控制就无法返回抛出点。

当异常发生时, 从异常点到异常处理程序是可以传递信息的。这些信息包括抛出对象的类型或抛出对象中的信息。

抛出的对象通常是个字符串 (错误消息) 或类对象。抛出对象将信息传递给处理该异常的异常处理程序。

**软件工程知识 13.4** 异常处理的关键是程序或系统中处理异常的部分可以与检测并产生异常的部分完成不同或有明显差异。

## 13.5 简单的异常处理例子: 除数为 0

以下而这个简单的异常处理为例。图 13.1 中的程序用 try, throw 和 catch 检测除数为 0 的异常情况, 表示并处理除数为 0 的异常。

```
1 //Fig.13.1: fig13_01.cpp
2 //A simple exception handling example.
3 //Checking for a divide-by-zero exception.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 //Class DivideByZeroException to be used in exception
11 //handling for throwing an exception on a division by zero.
```

```
12 class DivideByZeroException {
13 public:
14     DivideByZeroException()
15         : message( "attempted to divide by zero" ) {}
16     const char *what() const { return message; }
17 private:
18     const char *message;
19 };
20
21 //Definition of function quotient. Demonstrates throwing
22 //an exception when a divide-by-zero exception is encountered.
23 double quotient( int numerator, int denominator )
24 {
25     if ( denominator == 0 )
26         throw DivideByZeroException();
27
28     return static_cast< double > ( numerator ) /denominator;
29 }
30
31 //Driver program
32 int main()
33 {
34     int number1, number2;
35     double result;
36
37     cout << "Enter two integers (end-of-file to end): ";
38
39     while ( cin >> number1 >> number2 ) {
40
41         //the try block wraps the code that may throw an
42         //exception and the code that should not execute
43         //if an exception occurs
44         try {
45             result = quotient( number1, number2 );
46             cout << "The quotient is: " << result << endl;
47         }
48         catch ( DivideByZeroException ex ) { //exception handler
49             cout << "Exception occurred: " << ex.what() << '\n';
50         }
51
52         cout << "\nEnter two integers (end-of-file to end): ";
53     }
54
55     cout << endl;
56     return 0; //terminate normally
57 }
```

输出结果:

```
Enter two integers (end-of-file to end) : 100 7
The quotient is: 14.2857
```

```

Enter two integers (end-of-file to end): 100 0
Exception occurred; attempted to divide by zero

Enter two integers (end-of-file to end): 33 9
The quotient is: 3.66667

Enter two integers (end-of-file to end):

```

图 13.1 简单的异常处理例子:除数为 0

下面来看 main 中的驱动程序。注意 number1 和 number2 的局部声明。

程序包括了一个 try 程序块(见第 44 行),其中的代码可能抛出异常。注意,try 块中并没有显式列出可能造成错误的实际除法,而是通过 quotient 函数调用实际除法的代码。函数 quotient(定义在 23 行)实际抛出除数为 0 的异常对象,将在稍后介绍。一般说来,错误可能通过 try 块中的代码体现出来,可能通过函数调用体现出来,也可能通过 try 块的代码中启动的深层嵌套函数调用体现出来。

try 块后面是个 catch 块,包含除数为 0 的异常的异常处理程序。一般来说,try 块中抛出异常时,catch 块会捕捉这个异常,表明所抛出异常的相应类型。图 13.1 中,catch 块指定捕捉 DivideByZeroException 类型的异常对象,该对象类型符合函数 quotient 中抛出的异常对象类型。该异常处理程序输出函数 what 返回的一个错误消息。当然,真正的异常处理程序功能更为复杂。

执行时,如果 try 块中的代码没有抛出异常,则立即跳过 try 块后面的所有 catch 异常处理程序,执行 catch 异常处理程序后面的第一条语句,图 13.1 中执行 return 语句,返回 0 表示正常终止。

现在来看看 DivideByZeroException 类和 quotient 函数的定义。在函数 quotient 中,if 语句确定除数为 0 时,if 语句体发出一个 throw 语句,指定异常对象的构造函数名。这样就创建了 DivideByZeroException 的类对象。try 块后的 catch 语句(指定类型 DivideByZeroException)用于捕捉该对象。

DivideByZeroException 类的构造函数只是将 message 数据成员指向字符串“attempted to divide by zero”。catch 处理程序指定的参数(这里参数为 ex)中接收抛出的对象,并通过调用函数 what 打印这个消息。

**良好编程习惯 13.4** 将各种执行时错误与相应的命名异常对象关联,可使程序更清晰。

## 13.6 抛出异常

关键字 throw 表示有异常产生,称为抛出异常。throw 通常指定一个操作数(这里将介绍不指定操作数的特殊情况)。throw 的操作数可以是任何类型。如果操作数是对象,则称为异常对象。也可以不抛出对象,而是抛出条件表达式,也可能抛出不是用于错误处理的对象。

何处捕捉异常? 抛出异常时,指定相应类型的最近一个异常处理程序(对抛出该异常的 try 块而言)会捕捉这个异常。try 块的异常处理紧接在 try 块后面。

抛出异常时,会生成和初始化 throw 操作数的一个临时副本。然后这个临时对象初始化异常处理程序中的参数。异常处理程序执行完毕和退出时,删除临时对象。

**软件工程知识 13.5** 如果需要传递导致异常的错误信息,可以把此类信息放入抛出对象。catch 处理程序包含引用此类信息的参数名。

**软件工程知识 13.6** 抛出异常对象时也可以不传递信息,此时只需知道抛出这种类型的对象已提供了异常处理程序成功完成工作所需的足够信息。

抛出异常时,控制权退出当前 try 块,进入 try 块后面相应的 catch 处理器(如果有的话)。抛出点也可能在 try 块内多重的嵌套中,但控制仍将传入这个 catch 处理程序;抛出点也可能在多重的嵌套函数调用中,控制也会转入这个 catch 处理程序。

try 块可能不包含错误检查,也不包括 throw 语句,但 try 块中所引用的代码可能会执行其构造函数中的错误检查代码。try 块代码可能给数组类对象添上数组下标,其 operator[] 成员函数被重载“抛出”下标超界错误的异常。任何函数调用均可调用可能“抛出”异常的代码或另一个可能“抛出”异常的函数。

尽管异常可以终止程序执行,但不能一概而论。不过异常会终止在异常所在的程序块。

**常见编程错误 13.3** 异常只能在 try 块中抛出,如果在 try 块外部抛出异常,可能会调用 terminate。

**常见编程错误 13.4** 可能会抛出条件异常。但一定要小心使用,因为提升规则可能使条件表达式返回的值不是需要的类型。例如,从同一条件表达式抛出 int 或 double 时,条件表达式将 int 变成 double。因此,结果总是由参数为 double 的处理程序捕捉,而不是有时由参数为 double 的处理程序捕捉,有时由参数为 int 的处理程序捕捉。

## 13.7 捕捉异常

异常处理程序放在 catch 块中。每个 catch 块以关键字 catch 开始,接着是用括号括起来的类型(表示该 catch 块处理的异常类型)和可选参数名,然后是用花括号括起来的描述异常处理程序的代码。捕捉异常时,执行 catch 块中的代码。

catch 处理程序定义了自己的作用域。catch 在括号中指定要捕捉的对象类型。catch 处理程序中的参数可以命名也可以无名。如果是命名参数,则可以在处理程序中引用这个参数。如果是无名参数(只指定匹配抛出对象类型的类型),则信息不从抛出点传递到处理程序中,只是把控制从抛出点转入处理程序,许多异常都认可这样的处理。

**常见编程错误 13.5** 指定用逗号分开的 catch 参数表是语法错误。

抛出异常对象类型与 catch 处理程序参数类型匹配时,执行该 catch 块,即执行该类型的异常处理程序。

捕捉异常的 catch 处理程序是在当前活动 try 块后面第一个符合所抛出对象的处理程序,稍后将介绍匹配规则。

对于不能捕捉的异常,将调用 terminate(默认调用 abort)来终止程序。也可以指定自定

义的行为,即在 `set_terminate` 函数调用中指定函数名作为参数来执行该函数。

`catch` 后跟括号和省略号

```
catch(...)
```

表示捕捉所有异常。

**常见编程错误 13.6** 将 `catch(...)` 放在其他 `catch` 块之前时,其他块根本无法执行。`try` 块之后的处理程序列表中 `catch(...)` 总是最后一个。

**软件工程知识 13.7** 用 `catch(...)` 捕捉异常有两个缺点:其一是始终无法确定异常类型;其二是没有命名参数,就无法在异常处理程序中引用异常对象。

抛出的对象可能没有任何匹配的异常处理程序。这时,匹配搜索会延到外层的 `try` 块。这个过程将一直继续,如果最终找不到任何匹配的异常处理程序,就调用 `terminate` (默认调用 `abort`) 终止程序。

为寻找匹配项,会按顺序搜索异常处理程序,并执行第一个匹配的处理程序。处理程序执行完毕时,控制恢复到最后一个 `catch` 块后面的第一条语句,即该 `try` 块中最后一个异常处理程序后面的第一条语句。

抛出的对象可以有可能几个匹配的异常处理程序。这时将执行第一个匹配的异常处理程序。如果几个异常处理程序都匹配所抛出的对象,而每个异常处理程序用不同方法处理异常,则处理程序的顺序会影响异常的处理。

也可能几个 `catch` 处理程序中都包含可匹配所抛出对象的类型。原因有二:其一,有一个捕捉任何异常的 `catch(...)` 处理程序;其二,由于继承层次,派生类对象可以由派生类类型的异常处理程序捕捉,也可以由基类类型的异常处理程序捕捉。

**常见编程错误 13.7** 将对基类类型异常的“捕捉”放在对派生类类型的“捕捉”之前是逻辑错误。基类类型 `catch` 会捕捉所有从该类派生出来的所有对象,因此不会执行派生类类型的 `catch`。

**测试和调试提示 13.1** 程序员需要确定异常处理程序列出的顺序。这个顺序可能影响 `try` 块中所产生异常的处理方法。如果程序处理异常时出现意外行为,可能是前面的 `catch` 块捕获并处理了这个异常,使其没有被指定的异常处理程序处理。

有时,程序可能处理许多密切相关的异常类型。这时不是给每个不同异常提供不同的异常类和 `catch` 处理程序,而是给一组异常提供一个异常类和 `catch` 处理程序。每个异常产生时,可以创建具有不同 `private` 数据的异常对象。`catch` 处理程序通过检查 `private` 数据来区分异常的类型。

何时会有匹配呢? 下列情况下, `catch` 处理程序参数类型和所抛出对象的类型匹配:

- 实际是同一类型;
- `catch` 处理程序参数类型是所抛出对象类的 `public` 基类;
- 处理程序参数为基类指针或引用类型,而抛出对象为派生类指针或引用类型;
- `catch` 处理器为 `catch(...)`。

**常见编程错误 13.8** 将带 `void *` 参数类型的异常处理程序放在具有其他指针类型的异常处



理程序之前是逻辑错误。void \* 处理程序捕捉所有指针类型的异常,因此根本不可能执行其他异常处理程序。

准确的类型匹配是必须的。寻找处理程序时只允许派生类向基类转换,不允许其他转换和升级。

允许抛出常量对象。这时,catch 处理程序参数类型也被声明为 const。

如果异常没有相应的处理程序,程序就会终止。尽管这种作法正确,但不是程序员惯有的做法。错误是常有的事,但不会影响程序的继续执行。

后面跟有几个 catch 块的 try 块类似于 switch 语句。不必按照跳过其余异常处理程序的方式用 break 退出异常处理程序。每个 catch 块定义不同的作用范围,switch 语句则不同,所有条件都包含在 switch 的范围中。

**常见编程错误 13.9** 将分号放在 try 块之后或 try 块之后的任何 catch 处理程序(最后一个 catch 处理程序除外)之后是语法错误。

异常处理程序无法访问 try 块中定义的自动化对象,因为发生异常时 try 块被终止了,try 块中的所有自动对象均在处理程序开始执行之前被删除。

异常处理程序中发生异常时会出现什么情况呢?异常处理程序开始执行时,原先捕捉的异常正在处理。因此异常处理程序中发生异常时应在抛出原异常的 try 块之外处理。

异常处理程序可以用不同方式编写。可以检查错误和确定调用 terminate;可以再抛出异常(参见 13.8 节);可以将一种异常变为另一种异常,抛出另外的异常;可以进行必要的恢复,并恢复执行最后一个异常处理程序之后的语句;可以检查错误原因、删除错误和重新调用原先导致异常的函数(这不会生成无穷递归);可以向程序环境返回一些状态值等。

**软件工程知识 13.8** 最好在设计过程中把异常处理策略加入系统,因为在系统实施后再加入有效的异常处理策略很困难。

try 块不抛出任何异常而正常执行完毕时,控制传入 try 块后最后一个 catch 处理程序之后的第一条语句。

在 catch 处理程序中用 return 语句无法返回抛出点。这种 return 语句只返回调用 catch 块所在函数的函数。

**常见编程错误 13.10** 认为处理异常后控制会返回抛出点之后第一条语句是逻辑错误。

**软件工程知识 13.9** 传统控制流程不用异常的另一个原因是这些“额外增加的”异常可能打乱真正的错误型异常。程序员更难跟踪异常种类。例如,程序处理大量异常时,如何确定 catch(...)捕捉的异常呢?异常情况只能是较为少见的,不常发生的情况。

捕捉异常时,try 块中可能有已经分配但尚未释放的资源。如果可能,catch 处理程序应释放这些资源。例如,catch 处理程序删除通过 new 分配的空间和关闭抛出异常的 try 块中打开的任何文件。

catch 块处理错误之后可以让程序继续运行,也可以终止程序。

catch 处理程序本身也可以发现错误和抛出异常。这种异常不是由抛出异常的 catch 处理程序所在 try 块中的 catch 处理程序处理,而是由外层 try 块的 catch 处理程序处理。

**常见编程错误 13.11** catch 处理程序抛出的异常由该处理程序处理,或由抛出异常的 try 块中相关的处理程序(导致执行原先的 catch 处理程序)处理是逻辑错误。

## 13.8 重抛出异常

捕捉异常的处理程序也可以决定不处理异常或释放资源,而是让其他处理程序处理这个异常。这时,处理程序只须重抛出异常

```
throw;
```

这种不带参数的 throw 用于重抛出异常。如果开始不抛出异常,重抛出异常就要调用 terminate。

**常见编程错误 13.12** 将空 throw 语句放在 catch 处理程序之外;执行这样的 throw 操作将调用 terminate。

即使处理程序能处理异常,无论这个异常处理与否,处理程序仍然可以重抛出异常在处理程序之外继续处理。

重抛出的异常由外层 try 块检测,由该 try 块之后的异常处理程序处理。

**软件工程知识 13.10** 可以用 catch(...) 执行与异常类型无关的恢复,如释放共用资源。异常也可以重抛出以提醒更具体的外层 catch 块。

图 13.2 中的程序演示了重抛出异常。main 中第 31 行的 try 块中调用了函数 throwException。在函数 throwException 的 try 块中,第 17 行的 throw 语句抛出标准库类 exception 的实例(在头文件 <exception> 中定义)。这个异常在第 19 行的 catch 处理程序中立即捕捉,打印一条错误消息,然后重抛出异常。因此终止函数 throwException 并将控制权返回 main 中的 try/catch 块。第 34 行重新捕捉异常并打印一条错误消息。

```
1 //Fig.13.2: fig13_02.cpp
2 //Demonstration of rethrowing an exception.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <exception>
9
10 using std::exception;
11
12 void throwException()
13 {
14     //Throw an exception and immediately catch it.
15     try {
16         cout << "Function throwException\n";
17         throw exception(); //generate exception
18     }
19     catch( exception e )
20     {
```

```

21     cout << "Exception handled in function throwException\n";
22     throw; //rethrow exception for further processing
23 }
24
25     cout << "This also should not print\n";
26 }
27
28 int main()
29 {
30     try {
31         throwException();
32         cout << "This should not print\n";
33     }
34     catch ( exception e )
35     {
36         cout << "Exception handled in main\n";
37     }
38
39     cout << "Program control continues after catch in main"
40         << endl;
41     return 0;
42 }

```

输出结果:

```

Function throwException
Exception handled in function throwException
Exception handled in main
Program control continues after catch in main

```

图 13.2 重抛出异常

## 13.9 异常的规约

异常的规约

```

int g( double h ) throw (a, b,c)
{
    //function body
}

```

可以枚举指定函数所抛出的一系列异常。函数抛出的异常类型是可以限制的,函数声明中可以指定异常类型作为异常的规约(也称为抛出表“throwlist”),异常的规约列出了可抛出的异常。函数可以抛出指定异常类型或派生的类型。尽管这样似乎可保证不会抛出其他异常类型,事实却相反,仍然可以抛出其他异常类型。如果抛出异常规约中未列出的异常,会调用函数 unexpected。

将 throw() (即空异常规约)放在函数的参数表之后表示该函数不抛出任何异常。但这种函数仍然能抛出异常,而且也会调用函数 unexpected。

**常见编程错误 13.13** 如果抛出函数异常规约中未列出的异常,就会调用函数 unexpected。

### 不带异常规约的函数

```
void g(); //this function can throw any exception
```

可以抛出任何异常。函数 `unexpected` 的含义可以调用函数 `set_unexpected` 进行重新定义。

对于异常处理,有趣的一点是函数在 `throw` 表达式中包含函数异常指定中未列出的异常时,编译器不会产生语法错误。在捕捉该异常前,函数必须抛出这个异常。

如果函数抛出特定类类型的异常,那么它也可以抛出以 `public` 方式继承该类的所有派生类的异常。

## 13.10 处理意外异常

函数 `unexpected` 调用 `set_unexpected` 函数指定的函数。如果不指定函数,则默认调用 `terminate`。

函数 `terminate` 可以显式调用。在无法捕捉抛出的异常时、在异常处理期间堆栈被打乱时、作为调用 `unexpected` 的默认操作时和在异常导致堆栈解退的过程中析构函数试图抛出异常时,都会调用 `terminate`。

函数 `set_terminate` 可以指定调用 `terminate` 时调用的函数,否则 `terminate` 默认调用 `abort`。

函数 `set_terminate` 和 `set_unexpected` 的函数原型分别包含头文件 `<terminate.h>` 和 `<unexpected.h>` 中。

函数 `set_terminate` 和 `set_unexpected` 分别返回 `terminate` 和 `unexpected` 调用的最后一个函数的指针。这样一来程序员就可以保存函数指针,以便后期恢复。

函数 `set_terminate` 和 `set_unexpected` 取函数指针作为参数。每个参数必须指向返回类型为 `void` 和无参数的函数。

如果用户自定义终止函数的最后一个操作不是退出程序,那么在执行完用户自定义终止函数的其他语句之后,程序将自动调用函数 `abort` 终止程序。

## 13.11 堆栈解退

特定范围中异常已抛出但未被捕捉时,函数调用堆栈解退,尝试图外层 `try/catch` 块中捕捉这个异常。函数调用堆栈解退表示终止未捕捉异常的函数,删除函数的所有局部变量,控制返回函数调用点。如果函数调用点在 `try` 块中,则试图捕捉这个异常;如果函数调用点不在 `try` 块中或没有捕捉这个异常,则再次执行堆栈解退。正如前文所述,如果程序最终捕捉不到该异常,就会调用函数 `terminate` 终止程序。图 13.3 中的程序演示了堆栈解退。

```
1 //Fig. 13.3: fig13_03.cpp
2 //Demonstrating stack unwinding.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
```

```
8 #include <stdexcept>
9
10 using std::runtime_error;
11
12 void function3() throw ( runtime_error )
13 {
14     throw runtime_error( "runtime_error in function3" );
15 }
16
17 void function2() throw ( runtime_error )
18 {
19     function3();
20 }
21
22 void function1() throw ( runtime_error )
23 {
24     function2();
25 }
26
27 int main()
28 {
29     try {
30         function1();
31     }
32     catch ( runtime_error e )
33     {
34         cout << "Exception occurred: " << e.what() << endl;
35     }
36
37     return 0;
38 }
```

输出结果:

Exception occurred; runtime\_error in function3

图 13.3 堆栈解退的演示

main 中,第 30 行的 try 块调用 function1。接着 function1 (定义在第 22 行)调用 function2。随后 function2(定义在第 13 行)的调用 function3。function3 的第 14 行抛出 exception 对象。由于第 14 行不在 try 块中,因此发生堆栈解退,function3 终止,控制权返回 function2 (第 19 行)。由于第 19 行不在 try 块中,再次发生堆栈解退,function2 终止,控制权返回 function1(第 24 行)。由于第 24 行不在 try 块中,又一次发生堆栈解退,function1 终止,控制权返回 main(第 20 行)。由于第 30 行在 try 块中,可以捕捉异常,并在 try 块后面第一个匹配的 catch 处理程序中处理(第 32 行)。

## 13.12 构造函数、析构函数与异常处理

首先要处理前面已提到但尚未彻底解决的问题,即构造函数中发现错误时会发生什么

情况? 例如, `String` 构造函数在 `new` 失败, 无法取得保持 `String` 的内部表示所需空间时如何响应? 问题是构造函数无法返回数值, 如何让外部知道对象没有构造成功呢? 一种方案是返回没有正确构造的对象, 希望对象使用者通过相应测试来确定该对象是不能使用的对象。另一种方案是在构造函数外部设置一些变量。抛出的异常向外部传递失败的构造函数信息, 并负责处理这个失败。

为了捕捉异常, 异常处理程序必须访问所抛出对象的构造函数的部分(也可访问默认成员的部分)。

抛出异常之前, 构造函数抛出的异常会触发对要构造对象的析构函数调用。

抛出异常之前, 每个 `try` 块中构造的自动对象都调用析构函数。处理异常的同时, 处理器开始执行; 堆栈解退保证已经完成。如果堆栈解退所调用的析构函数抛出异常, 则调用 `terminate`。

如果对象有成员函数, 且如果在外层对象构造完成之前有异常抛出, 则在发生异常之前, 执行构造成员对象的析构函数。

如果异常发生时对象数组被部分构造, 则只调用已构造的数组元素的析构函数。

异常可能跳过通常释放资源的代码, 从而造成资源泄漏。要解决这个问题, 一种方法是在请求资源时初始化一个局部对象。发生异常时, 调用析构函数并释放资源。

要捕捉析构函数抛出的异常, 可以将调用析构函数的函数放入 `try` 块, 并提供相应类型的 `catch` 处理程序。抛出对象的析构函数在异常处理程序执行完毕之后执行。

## 13.13 异常与继承

从 `public` 基类可以派生各种异常类。如果 `catch` 捕捉了一个指向基类类型异常对象的指针或引用, 那么它也可以捕捉该基类所派生的异常对象的指针或引用。相关错误的多态处理是允许的。

**测试和调试提示 13.2** 利用异常继承可使异常处理程序用相当简单的符号捕捉相关错误。虽然可以捕捉每个派生类异常对象的指针或引用, 但捕捉基类异常对象的指针或引用更为简练。而且, 如果程序员忘记显式测试一个或几个派生类指针或引用, 捕捉每个派生类异常对象的指针或引用的方法就容易造成错误。

## 13.14 处理 `new` 失败

处理 `new` 失败的方法有多种。我们介绍过用宏 `assert` 测试 `new` 返回的值。如果返回值为 0, 宏 `assert` 就会终止程序。这不是处理 `new` 失败的健壮机制, 它不允许我们用任何方法从失败中恢复。C++ 标准指定, 出现 `new` 失败时, 将抛出 `bad_alloc` 异常(在头文件 `<new>` 中定义)。但许多编译器目前还不支持该标准, 仍然会在 `new` 失败时返回 0。本节介绍 3 个 `new` 失败的例子。第一个例子在 `new` 失败时返回 0。后两个例子在出现 `new` 失败时, 抛出 `bad_alloc` 异常。

图 13.4 演示了 `new` 请求分配内存失败时返回 0。第 12 行的 `for` 结构循环 50 次, 每次循

环分配 5 000 000 个 double 值的数组(即 40 000 000 字节,因为 double 通常为 8 个字节)。第 15 行的 if 结构测试每次 new 操作中是否分配内存成功。如果 new 请求内存失败返回了 0,就会打印“Memory allocation failed”消息,循环终止。

```

1 //Fig.13.4; fig13_04.cpp
2 //Demonstrating new returning 0
3 //when memory is not allocated
4 #include <iostream>
5
6 using std::cout;
7
8 int main()
9 {
10     double *ptr[ 50 ];
11
12     for ( int i = 0; i < 50; i++ ) {
13         ptr[ i ] = new double[ 5000000 ];
14
15         if ( ptr[ i ] == 0 ) { //new failed to allocate memory
16             cout << "Memory allocation failed for ptr[ "
17                 << i << " ]\n";
18             break;
19         }
20         else
21             cout << "Allocated 5000000 doubles in ptr[ "
22                 << i << " ]\n";
23     }
24
25     return 0;
26 }
```

输出结果:

```

Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Memory allocation failed for ptr[ 4 ]
```

图 13.4 new 失败返回 0 的演示

输出表示 new 失败和循环终止之前只进行了 4 次循环。不同系统中的输出可能不同,取决于实际内存、可用的虚拟内存的磁盘空间和用于编译程序的编译器。

图 13.5 演示了 new 请求内存失败时抛出 bad\_alloc。第 18 行的 for 结构循环重 50 次,每次循环分配 5 000 000 个 double 值的数组(即 40 000 000 字节,因为 double 通常为 8 个字节)。如果 new 失败并抛出 bad\_alloc 异常,则循环终止,程序继续第 24 行的异常处理控制流程,捕捉和处理异常,打印“Exception occurred:”消息,然后 exception.what() 返回异常特定消息的字符串(对于 bad\_alloc 为“Allocation Failure”)。输出表示 new 失败和抛出 bad\_alloc 异常之前只进行了 4 次循环。不同系统的输出可能不同,具体情况与实际内存、可用虚

拟内存的磁盘空间和用于编译程序的编译器有关。

```

1  //Fig. 13.5: fig13_05.cpp
2  //Demonstrating new throwing bad_alloc
3  //when memory is not allocated
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <new>
10
11 using std::bad_alloc;
12
13 int main()
14 {
15     double *ptr[ 50 ];
16
17     try {
18         for ( int i = 0; i < 50; i++ ) {
19             ptr[ i ] = new double[ 5000000 ];
20             cout << "Allocated 5000000 doubles in ptr[ "
21                 << i << " ]\n";
22         }
23     }
24     catch ( bad_alloc exception ) {
25         cout << "Exception occurred: "
26             << exception.what() << endl;
27     }
28
29     return 0;
30 }

```

输出结果:

```

Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Exception occurred: Allocation Failure

```

图 13.5 new 失败抛出 bad\_alloc 异常

编译器对 new 失败处理的支持各不相同。许多 C++ 编译器在 new 失败时默认返回 0。有些编译器在包括头文件 <new> (或 <new.h>) 时支持 new 抛出异常。有些编译器不管是否包括头文件 <new> 都默认抛出 bad\_alloc。

C++ 标准指定标准支持的编译器在 new 失败时仍然可以用返回 0 的版本。为此,头文件 <new> 定义类型 nothrow, 使用语句

```
double *ptr = new (nothrow) double [ 5000000 ];
```

该语句表示用不支持抛出 bad\_alloc 异常的 new 版本 (即 nothrow) 分配 5 000 000 个 double



值的数组。

**软件工程知识 13.11** 为了使程序更健壮,C++ 标准建议程序员使用抛出 `bad_alloc` 异常的 `new` 版本。

还可以用另一个特性处理 `new` 失败。函数 `set_new_handler` (原型在头文件 `<new>` 或 `<new.h>` 中)取一个函数指针作为参数,所指函数不带参数并返回 `void`。函数指针注册为 `new` 失败时要调用的函数。这样便为程序员提供了处理每个 `new` 失败的通用方法,而不管失败发生在程序中哪个地方。程序中,用 `set_new_handler` 注册 `new` 处理程序之后,`new` 不会在失败时抛出 `bad_alloc`。

操作符 `new` 实际上是一个循环,请求所要的内存。如果内存分配成功,`new` 就返回指向该内存的指针。如果内存分配失败,且没有用 `set_new_handler` 注册 `new` 处理程序函数,则 `new` 抛出 `bad_alloc` 异常。如果 `new` 无法分配内存而注册了 `new` 处理程序函数,则调用 `new` 处理程序函数。C++ 标准指定 `new` 处理程序函数应完成下列任务:

(1)通过删除其他动态分配内存获得更多的内存空间,然后返回 `new` 操作符中的循环,试图再次分配内存;

(2)抛出 `bad_alloc` 类型的异常;

(3)调用函数 `abort` 或 `exit` (都在 `<cstdlib>` 头文件中定义)终止程序。

图 13.6 中的程序演示了 `set_new_handler`。函数 `customNewHandler` 只是打印错误消息和调用 `abort` 终止程序。输出显示 `new` 失败和抛出 `bad_alloc` 异常之前循环进行了 3 次迭代。不同系统的输出结果可能不同,具体情况与取决于实际内存、可用的虚拟内存的磁盘空间和编译程序的编译器有关。

```

1 //Fig.13.6: fig13_06.cpp
2 //Demonstrating set_new_handler
3 #include <iostream>
4
5 using std::cout;
6 using std::cerr;
7
8 #include <new>
9 #include <cstdlib>
10
11 using std::set_new_handler;
12
13 void customNewHandler()
14 {
15     cerr << "customNewHandler was called";
16     abort();
17 }
18
19 int main()
20 {
21     double *ptr[ 50 ];
22     set_new_handler( customNewHandler );
23 
```

```

24     for ( int i = 0; i < 50; i++ ) |
25         ptr[ i ] = new double[ 5000000 ];
26
27         cout << "Allocated 5000000 doubles in ptr[ "
28             << i << " ]\n";
29     }
30
31     return 0;
32 |

```

输出结果:

```

Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
coustonNewHandler was called

```

图 13.6 演示 set\_new\_handler

## 13.15 auto\_ptr 类与动态内存分配

在闲置存储区分配动态内存(可能是对象),将该内存的地址赋给一个指针,利用指针来操作内存,并在不再需要该内存时用 delete 释放内存是一个较为常见的编程习惯。如果内存分配之后和执行 delete 语句之前发生异常,可能会造成发生内存泄漏。C++ 标准提供 <memory> 头文件中的 auto\_ptr 类模板来解决这一问题。

auto\_ptr 类对象维护动态分配内存的指针。auto\_ptr 对象超出范围时,就对指针数据成员执行 delete 操作。auto\_ptr 类模板提供了 \* 和 -> 操作符,因此可以像使用普通指针变量一样使用 auto\_ptr 对象。图 13.7 演示了 auto\_ptr 对象指向 Integer 类对象(定义在第 12 ~ 第 22 行)。

```

1  //Fig. 13.7: fig13_07.cpp
2  //Demonstrating auto_ptr
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <memory>
9
10 using std::auto_ptr;
11
12 class Integer |
13 public:
14     Integer( int i = 0 ) : value( i )
15         | cout << "Constructor for Integer " << value << endl; }
16     ~Integer()
17         | cout << "Destructor for Integer " << value << endl; }
18     void setInteger( int i ) { value = i; |

```

```

19     int getInteger() const { return value; }
20 private:
21     int value;
22 };
23
24 int main()
25 {
26     cout << "Creating an auto_ptr object that points "
27         << "to an Integer\n";
28
29     auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
30
31     cout << "Using the auto_ptr to manipulate the Integer\n";
32     ptrToInteger -> setInteger( 99 );
33     cout << "Integer after setInteger: "
34         << ( *ptrToInteger ).getInteger()
35         << "\nTerminating program" << endl;
36
37     return 0;
38 }

```

输出结果:

```

Creating an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99
Terminating program
Destructor for Integer 99

```

图 13.7 演示 auto\_ptr 类

第 29 行

```
auto_ptr<integer>ptrToInteger(new integer(7));
```

创建了 auto\_ptr 对象 ptrToInteger,并将其初始化为动态分配 Integer 对象的指针(包含数值 7)。

第 32 行

```
ptrToInteger->setInteger(99);
```

用 auto\_ptr 重载 -> 操作符和函数调用操作符()调用 ptrToInteger 所指 Integer 对象的 setInteger 函数。

第 34 行

```
( *ptrToInteger ).getInteger()
```

用 auto\_ptr 重载 \* 操作符复引用 ptrToInteger,然后用圆点操作符(.)和函数调用操作符()调用 ptrToInteger 所指 Integer 对象的 getInteger 函数。

由于 ptrToInteger 是 main 的局部自动变量,因此 main 终止时会删除 ptrToInteger。这样便可强制删除 ptrToInteger 所指的 Integer 对象,从而强制调用 Integer 类的析构函数。更重要的是这个方式可有效防止内存泄漏。

## 13.16 标准库异常的层次结构

经验表明,异常是可以分类的。C++ 标准提供了标准库异常的层次结构。这个层次结构以基类 `exception` 开头(在头文件 `<exception>` 中定义),该基类提供了函数 `what()`,在每个派生类中重定义发出相应的错误消息。

可从基类 `exception` 派生直接派生类 `runtime_error` 和 `logic_error`(均在头文件 `<stdexcept>` 中定义),每个派生类又可以派生其他类。

还可从基类 `exception` 派生因 C++ 语言特性而抛出的异常,例如, `new` 抛出 `bad_alloc`(参见 13.14 节), `dynamic_cast` 抛出 `bad_cast`(参见第 21 章), `typeid` 抛出 `bad_typeid`(参见第 21 章)。如果发生意外异常,在函数的抛出表中加上 `std::bad_exception`, `unexpected()` 抛出 `bad_exception` 而不是(默认)终止程序或调用 `set_unexpected` 指定的函数。

`logic_error` 类是几个标准异常类的基类,表示程序逻辑错误,这些错误可以通过编写正确的代码来避免。下面介绍其中的一些类。`invalid_argument` 类表示向函数传入无效参数(精心编写正确的代码可避免这一错误)。`length_error` 类表示长度大于所操作对象允许的最大长度(第 19 章处理 `string` 时会抛出 `length_error` 异常)。`out_of_range` 类表示数组和 `string` 下标之类的值超界。

`runtime_error` 类是其他几个标准异常类的基类,表明只能在执行时发现的程序中的错误。`overflow_error` 类表示发生运算上溢错误;`underflow_error` 类表示发生运算下溢错误。

**软件工程知识 13.12** 标准的 `exception` 层次结构只是一个起点,用户可以抛出标准异常、抛出从标准异常派生的异常或抛出非标准异常派生的异常。

**常见编程错误 13.14** 用户自定义的异常类无须从 `exception` 类派生。所以编写 `catch(exception e)` 并不能保证捕捉程序可能遇到的所有异常。

**测试和调试提示 13.3** 要捕捉 `try` 块可能抛出的所有异常,应用 `catch(...)`。

## 13.17 小结

- 异常的常见例子有数组下标超界、运算溢出、除数为 0、无效函数参数和 `new` 无法取得所需内存。
- 异常处理使程序可以捕捉和处理错误,而不是任其发生和造成恶果。如果程序员不提供处理致命错误的过程,则程序终止;虽然非致命错误通常允许程序继续执行,但会产生错误结果。
- 异常处理设计用于处理同步错误,作为程序执行的结果。
- 异常处理不能用于处理异步情况,如磁盘 I/O 完成、网络消息到达、鼠标单击等等,这些情况最好用其他方法处理,如中断处理。
- 异常处理通常用于发现错误部分与处理错误部分处于不同位置(不同范围)时。
- 异常不应用作为具体的控制流机制。传统控制结构的控制流一般比异常更清晰、更

高效。

- 异常处理应当用于处理程序组件中与这些异常处理没有直接关系的异常。
- 异常处理应当用于处理函数、库、类等常用软件组件中的异常和组件本身不处理异常时。
- 异常处理应当用于在大型项目中以统一方式处理整个项目异常。
- C++ 异常处理用于错误检测函数无法处理错误的情况。这种函数抛出异常。如果异常与 catch 块中的参数类型相符,就执行该 catch 块的代码。如果找不到匹配的异常处理程序,就调用函数 `terminate`(默认调用函数 `abort`)。
- 程序员在 try 块中放入出错时产生异常的代码。try 块后面是一个或几个 catch 块。每个 catch 块指定捕捉和处理一种异常。每个 catch 块包含一个异常处理程序。
- 抛出异常时,程序控制离开 try 块,从 catch 块中搜索相应的异常处理程序。如果 try 块中没有抛出异常,则跳过该块的异常处理程序,程序在最后一个 catch 块之后恢复执行。
- 函数的 try 块中抛出异常,或者从 try 块直接或间接调用的函数抛出异常。
- 抛出异常之后,控制权无法返回抛出点。
- 发生异常时,可以从异常点向异常处理程序传递信息。这些信息是抛出对象的类型或抛出对象中的信息。
- 常见异常类型是 `char *`,该类型只包括一条错误消息,作为 `throw` 的操作数。
- 异常指定可以由指定函数抛出一系列类异常,将空异常指定语句放在函数的参数表之后表示该函数不抛出任何异常。
- 抛出异常时,指定相应类型的最近一个异常处理程序(对抛出该异常的 try 块)捕捉这个异常。
- 抛出异常时,生成和初始化 `throw` 操作数的一个临时副本,然后这个临时对象初始化异常处理程序中的参数。异常处理程序执行完毕和退出时,删除临时对象。
- 不一定总是显式检查错误。try 块可能不包含错误检查和 `throw` 语句,但 try 块中所指的代码可能导致执行构造函数中的错误检查代码。
- 异常会终止异常所在的程序块。
- 异常处理程序放在 catch 块中。每个 catch 块以关键字 `catch` 开始,接着是括号内的包含类型(表示该块处理的异常类型)和可选参数名。后面是用花括号括起来的描述异常处理程序的代码。
- 捕捉异常时,执行 catch 块中的代码。
- catch 处理程序定义自己的范围。
- catch 处理程序中的参数可以命名也可以无名。如果是命名参数,则可以在处理程序中引用这个参数,如果是无名参数(只指定匹配抛出对象类型的类型或用省略号表示所有类型),处理程序会忽略所有抛出异常。处理程序可以将对象重新抛出到外层 try 块。
- 可以指定自定义行为,在 `set _ minate` 函数调用中指定函数名参数,指定执行另一个函数来替代函数 `terminate`。

- `catch(...)` 表示捕捉所有异常。
- 也许某个抛出对象没有任何匹配的异常处理程序。这时匹配搜索会继续到外面一层 `try` 块。异常处理程序按顺序搜索, 寻找匹配项, 并执行第一个匹配的处理程序。处理程序执行完毕时, 控制恢复到最后一个 `catch` 块后面的第一条语句。
- 处理程序的顺序会影响处理异常的方法。
- 派生类对象可以由派生类类型的异常处理程序和基类类型的异常处理程序捕捉。
- 有时程序可能处理许多密切相关的异常类型。这时不是给每个不同异常提供不同的异常类和 `catch` 处理程序, 而是给一组异常提供一个异常类和 `catch` 处理程序。发生每个异常时, 可以生成具有不同 `private` 数据的异常对象。 `catch` 处理程序通过检查 `private` 数据来区分异常的类型。
- 即使有准确匹配, 也应在匹配时要求标准转换, 因为这个处理程序出现在触发准确匹配的处理程序之前。
- 默认情况下, 如果找不到异常的处理程序, 程序就会终止。
- 异常处理程序无法直接访问 `try` 块范围中的变量。处理器所需的信息通常通过抛出对象传递。
- 异常处理程序可以用不同方式编写, 可以检查错误和确定调用 `terminate`; 可以再抛出; 通过抛出不同异常可以将一种异常变为另一种异常, 可以进行必要的恢复, 并恢复执行最后一个异常处理程序之后的语句; 可以检查错误原因, 删除错误原因和重新调用原先导致异常的函数(这不会生成无穷递归); 可以向运行环境返回一些状态值等等。
- 应当将捕捉基类类型的异常处理程序放在捕捉派生类类型的异常处理程序之后, 否则基类类型的异常处理程序捕捉基类对象和从该类派生的所有对象。
- 捕捉异常时, `try` 块中可能有已分配但还没有释放的资源。 `catch` 处理程序应释放这些资源。
- 捕捉异常的处理程序也可以决定不处理异常。这时, 处理程序只须重抛出该异常。这种不带参数的 `throw` 重抛出异常。如果开始没有抛出异常, 则重抛出异常调用 `terminate`。
- 即使处理程序能处理异常, 不管这个异常是否进行处理, 处理程序仍然可以重抛出异常以便在处理程序之外继续处理。重抛出异常由外层 `try` 块检测, 由所在即块之后列出的异常处理程序处理。
- 不带异常指定的函数可抛出任何异常。
- 函数 `unexpected` 调用 `set_unexpected` 函数指定的函数。如果不用 `set_unexpected` 函数指定函数, 则默认调用 `terminate`。
- 函数 `terminate` 可以显式调用, 在无法捕捉抛出的异常时、在异常处理期间打乱堆栈时、作为调用 `unexpected` 的默认操作时或在异常导致堆栈解退调用析构函数抛出异常的情况下, 都会调用 `terminate`。
- 函数 `set_terminate` 和 `set_unexpected` 的原型分别包含在头文件 `<terminate.h>` 和 `<unexpected.h>` 中。

- 函数 `set_terminate` 和 `set_unexpected` 分别返回 `terminate` 和 `unexpected` 调用的最后一个函数的指针。如此一来,程序员可保存函数指针,以便后期恢复。
- 函数 `set_terminate` 和 `set_unexpected` 取函数指针为参数。每个参数指向返回类型为 `void` 和无参数的函数。
- 如果用户自定义终止函数的最后一个操作不是退出程序,则执行用户自定义终止函数的其他语句之后自动调用 `abort` 函数终止程序。
- `try` 块之外抛出的异常会使程序终止。
- 如果 `try` 块之后找不到处理程序,则继续堆栈解退,直到找到相应处理程序为止。如果最终找不到处理程序,则调用 `terminate` (默认用 `abort`) 退出程序。
- 异常指定列出可抛出的异常。函数可以抛出指定异常或派生类型。如果抛出异常指定中没有指定的异常,则调用函数 `unexpected`。
- 如果函数抛出特定类类型的异常,则函数也可以抛出用 `public` 继承从该类派生的所有类异常。
- 要捕捉异常,异常处理程序要访问所抛出对象的构造函数副本。
- 构造函数中抛出异常时,对所有已构造的基类对象和抛出异常之前构造的成员对象调用析构函数。
- 如果发生异常时部分构造了对象数组,则只调用已构造数组元素的析构函数。要捕捉析构函数中抛出的异常,可以将调用析构函数的函数放在 `try` 块中,并提供相应类型的 `catch` 处理程序。
- 利用异常继承使异常处理程序可以用相当简单的符号捕捉相关错误。虽然可以捕捉每个派生类的异常对象,但捕捉基类的异常对象的方法更为简练。
- C++ 标准指定,出现 `new` 失败时抛出 `bad_alloc` 异常(在头文件 `<new>` 中定义)。
- 许多编译器目前还不支持标准,仍然在 `new` 失败时返回 0。
- 函数 `set_new_handler` (原型在头文件 `<new>` 或 `<new.h>` 中) 取一个函数指针参数,所指函数不取参数并返回 `void`。函数指针注册为 `new` 失败时要调用的函数。用 `set_new_handler` 注册 `new` 处理程序之后,`new` 不会在发生失败时抛出 `bad_alloc`。
- 类 `auto_ptr` 对象维护动态分配内存的指针。当 `auto_ptr` 对象超出范围时,对指针数据成员进行一个 `delete` 操作。`auto_ptr` 类模板提供了 `*` 和 `->` 操作符,所以 `auto_ptr` 对象可以用作普通指针变量。
- C++ 标准提供了标准库异常层次。这个层次以基类 `exception` 开始(在头文件 `<exception>` 中定义),该基类提供服务 `what()`,在每个派生类中重定义,发出相应错误消息。
- 发生意外异常时,通过在函数的抛出表中加上 `std::bad_exception, unexpected()` 将抛出 `bad_exception` 而不是(默认)终止程序或调用 `set_unexpected` 指定的另一函数。

## 本章术语

`<exception>` header file 头文件 `<exception>`

`<memory>` headerfile 头文件 `<memory>`

|                               |             |                                          |           |
|-------------------------------|-------------|------------------------------------------|-----------|
| < new > headerfile            | < new > 头文件 | fault tolerance                          | 容错性       |
| assert macro                  | 宏 assert    | function with no exception specification | 不带异常指定的函数 |
| catch a group of exceptions   | 捕捉一组异常      | handle an exception                      | 处理异常      |
| catch an exception            | 捕捉一个异常      | handler for a baseclass                  | 基类处理程序    |
| catch argument catch          | 参数          | handler for a derived class              | 派生类处理程序   |
| catch block catch             | 块           | mission-critical application             | 任务关键的应用   |
| empty exception specification | 空异常指定       | nested exception handlers                | 嵌套的异常处理程序 |
| empty throw specification     | 空 throw 指定  | rethrow an exception                     | 重抛出异常     |
| enclosing try block           | 所在 try 块    | robustness                               | 健壮性       |
| exception declaration         | 异常声明        | stack unwinding                          | 堆栈解退      |
| exception handler             | 异常处理程序      | throw point                              | 抛出点       |
| exception list                | 异常表         | throw without arguments                  | 不带参数抛出    |
| exception object              | 异常对象        | try block                                | try 块     |
| exception specification       | 异常指定        | uncaught exception                       | 未捕捉到的异常   |
| exceptional condition         | 异常条件        |                                          |           |
| exception                     | 异常          |                                          |           |

### 常见编程错误

- 13.1 传统程序控制使用异常处理的另一危险是堆栈解退,异常发生之前分配的资源可能无法释放。这个问题可以通过认真编程来避免。
- 13.2 退出程序会使其他程序无法使用其资源,造成资源泄漏。
- 13.3 异常只能在 try 块中抛出,如果在 try 块外抛出异常,可能会调用 terminate。
- 13.4 可能会抛出条件异常。但一定要小心使用,因为提升规则可能使条件表达式返回的值不是自己需要的类型。例如,从同一条件表达式抛出 int 或 double 时,条件表达式将 int 变成 double。因此,结果总是由参数为 double 的处理程序捕捉,而不是有时由参数为 double 的处理程序捕捉,有时由参数为 int 的处理程序捕捉。
- 13.5 指定用逗号分开的 catch 参数表是语法错误。
- 13.6 将 catch(...) 放在其他 catch 块之前时,其他块根本无法执行。try 块之后的处理程序列表中,catch(...) 总是最后一个。
- 13.7 将对基类类型异常的“捕捉”放在对派生类类型的“捕捉”之前是逻辑错误。基类类型 catch 会捕捉所有从该类派生出来的所有对象,因此不会执行派生类类型的 catch。
- 13.8 将带 void \* 参数类型的异常处理程序放在具有其他指针类型的异常处理程序前面是逻辑错误。void \* 处理程序捕捉所有指针类型的异常,因此根本不可能执行其他异常处理程序。
- 13.9 将分号放在 try 块之后或 try 块之后的任何 catch 处理程序(最后一个 catch 处理程序除外)之后是语法错误。
- 13.10 认为处理异常后控制会返回 throw 之后第一条语句是逻辑错误。
- 13.11 catch 处理程序抛出的异常由该处理程序处理,或由抛出异常的 try 块中相关的处理程序(导致执行原先的 catch 处理程序)处理是逻辑错误。
- 13.12 将空 throw 语句放在 catch 处理程序之外:执行这样的 throw 操作将调用 terminate。
- 13.13 如果抛出的异常未在函数的异常规约中列出,就调用函数 unexpected。
- 13.14 用户自定义的异常类无须从 exception 类派生。所以 catch(exception e) 并不能保证捕



捉程序可能遇到的所有异常。

## 良好编程习惯

- 13.1 发生范围与处理范围不同的错误可以使用异常处理。发生范围与处理范围相同的错误,则应该使用其他方法处理。
- 13.2 使用异常处理时,尽量避免错误处理以外的其他用途的处理,这样可使程序更清晰。
- 13.3 对程序本身容易处理的简单而又局部性的错误,应用传统错误处理方法而避免使用异常处理。
- 13.4 将各种执行时错误与相应的命名异常对象关联,可使程序更清晰。

## 性能提示

- 13.1 尽管异常处理可以进行错误处理以外的工作,但将两种用途混用会降低程序性能。
- 13.2 编译器实现异常处理时,通常情况下,在异常不发生时将异常处理代码开销降到极小或为0,而发生异常时开出有关的系统开销。当然,异常处理代码的存在无疑会使程序占用更多内存。

## 软件工程知识

- 13.1 传统控制结构的控制流通常比使用了异常的控制流更清晰、更有效。
- 13.2 异常处理适用于分组件开发的系统。异常处理使组件组合更为容易。每个组件可以独立于其他异常处理范围之外,执行自己的异常检测。
- 13.3 对于库操作,库函数调用者通常用特定错误处理方法处理库函数中产生的异常。库函数很难进行可满足用户独特需求的错误处理。所以我们说,异常适合处理库函数产生的错误。
- 13.4 异常处理的关键是程序或系统中处理异常的部分可以和检测并产生异常的部分完成不同或有明显差异。
- 13.5 如果需要传递导致异常的错误信息,可以把此类信息放入抛出对象。catch 处理程序包含引用此类信息的参数名。
- 13.6 抛出异常对象时也可以不传递信息;此时只需知道抛出这种类型的对象已提供了异常处理程序成功完成工作所需的足够信息。
- 13.7 用 catch(...) 捕捉异常有两个缺点:其一是始终无法确定异常类型;其二是没有命名参数,就无法在异常处理程序中引用异常对象。
- 13.8 最好在设计过程中把异常处理策略加入系统,因为在系统实施后再加入有效的异常处理策略很困难。
- 13.9 传统控制流程不用异常的另一个原因是这些“额外的”异常可能打乱真正的错误型异常。程序员更难跟踪异常种类。例如,程序处理大量异常时,如何确定 catch(...) 捕捉的异常呢? 异常情况只能是较为少见的、不常发生的情况。
- 13.10 可以用 catch(...) 进行与异常类型无关的恢复,如释放共用资源。异常也可以重抛出以警示更具体的外层 catch 块。
- 13.11 为了使程序更健壮, C++ 标准建议程序员使用抛出 bad\_alloc 异常的 new 版本。
- 13.12 标准的 exception 层次结构只是一个起点,用户可以抛出标准异常、抛出从标准异常派

生的异常或抛出非标准异常派生的异常。

### 测试和调试提示

- 13.1 程序员需要确定异常处理程序列出的顺序。这个顺序可能影响 try 块中所产生异常的处理方法。如果程序处理异常时出现意外行为,可能是前面的 catch 块捕获并处理了这个异常,使其没有被指定的异常处理程序处理。
- 13.2 利用异常继承可使异常处理程序用相当简单的符号捕捉相关错误。虽然可以捕捉每个派生类异常对象的指针或引用,但捕捉基类异常对象的指针或引用更为简练。而且,如果程序员忘记显性地测试一个或几个派生类指针或引用,捕捉每个派生类异常对象的指针或引用的方法就容易造成错误。
- 13.3 要捕捉 try 块可能抛出的所有异常,应用 catch(... )。

### 自测题

- 13.1 列出 5 个常见的异常例子。
- 13.2 说明异常处理方法不能用于传统程序控制的原因。
- 13.3 为什么异常适用于处理库函数产生的错误?
- 13.4 什么是“资源泄漏”?
- 13.5 如果 try 块中不抛出异常,那么 try 块执行完毕之后控制权会转到何处?
- 13.6 如果在 try 块之外抛出异常,会发生什么情况?
- 13.7 说明使用 catch(... )的主要优点和主要缺点。
- 13.8 如果没有匹配抛出对象类型的 catch 处理程序,会发生什么情况?
- 13.9 如果有多个匹配抛出对象类型的 catch 处理程序,会发生什么情况?
- 13.10 为什么程序员要指定基类类型为 catch 处理程序类型,然后抛出派生类类型的对象?
- 13.11 catch 处理程序如何编写成处理相关错误类型而不用异常类之间的继承?
- 13.12 catch 处理程序中用哪种指针类型捕捉任何指针类型的所有异常?
- 13.13 假设有准确匹配异常对象类型的 catch 处理程序,什么情况下该异常对象类型会执行不同的 catch 处理程序?
- 13.14 抛出异常是否一定会终止程序?
- 13.15 catch 处理程序抛出异常时会发生什么情况?
- 13.16 throw; 语句有何用途?
- 13.17 程序员如何限制函数可以抛出的异常类型?
- 13.18 如果函数抛出函数异常指定中未列出的异常类型,会发生什么情况?
- 13.19 try 块抛出异常时,其中已经构造的自动对象发生什么情况?

### 自测题答案

- 13.1 new 无法取得所需内存、数组下标超界、运算溢出、除数为 0、无效函数参数。
- 13.2 a) 异常处理用于处理不常发生的情况,这些情况往往导致程序终止。所以,C++ 编译器的编写人员不必为实现最优化性能为实施异常处理。  
b) 使用传统控制结构的控制流通常比使用异常的控制流更清晰、更有效。  
c) 另一危险是堆栈解退,异常发生之前分配的资源可能无法释放。

- d) 这些“额外的”异常可能打乱真正的错误类型异常。程序员更难跟踪异常种类。例如,程序处理大量异常时,无法确定 `catch(...)` 捕捉的确定异常。
- 13.3 库函数很难执行可满足所有用户特殊需求的错误处理。
  - 13.4 退出程序会使其他程序无法使用其资源,从而造成资源泄漏。
  - 13.5 忽略 `try` 的异常处理程序(`catch` 块中),程序在最后一个 `catch` 块后重新执行。
  - 13.6 `try` 块外抛出的异常会使程序终止。
  - 13.7 `catch(...)` 能捕捉 `try` 块中抛出的所有错误。其优点是可以捕捉所有错误,缺点是 `catch` 没有参数,无法引用抛出的所有错误中的信息,无法知道错误原因。
  - 13.8 这会导致匹配搜索延伸到外层 `try` 块。这个过程会一直继续,也许最终找不任何匹配的异常处理程序。这时调用 `terminate`(默认调用 `abort` 终止程序)。可用 `set_terminate` 作为参数提供另一个 `terminate` 函数。
  - 13.9 执行 `try` 块后第一个匹配的异常处理程序。
  - 13.10 这是捕捉相关类型异常的好方法。
  - 13.11 可以用一个异常类和 `catch` 处理程序处理一组异常。发生每个异常时,可以生成具有不同 `private` 数据的异常对象。`catch` 处理程序通过检查 `private` 数据区分异常的类型。
  - 13.12 `void *`
  - 13.13 要求标准转换的处理程序可能出现在可精确匹配的处理程序之前。
  - 13.14 不一定,但它的确会终止抛出异常的块。
  - 13.15 异常由导致异常的 `catch` 处理程序所在 `try` 块(如果有的话)的相关 `catch` 处理程序(如果有)处理。
  - 13.16 重抛出异常。
  - 13.17 提供从函数可抛出异常类型的异常指定表。
  - 13.18 调用函数 `unexpected`。
  - 13.19 通过堆栈解退过程,调用每个对象的析构函数。

## 练习题

- 13.20 列举文中提及的程序中发生的各种异常条件,数量不限。对每个异常条件,简单描述程序如何用本章介绍的异常处理方法处理异常。典型的异常有:`new` 无法取得所需内存、数组下标超界、运算溢出、除数为0、无效函数参数。
- 13.21 什么情况下,程序员可在定义处理程序捕捉对象类型时不提供参数名?
- 13.22 语句
 

```
throw;
```

 通常出现在什么地方? 出现在别的地方会发生什么情况?
- 13.23 语句
 

```
catch(...) {throw};
```

 适用于哪些情况?
- 13.24 比较异常处理与各种其他错误处理方法。
- 13.25 列举异常处理相对于传统错误处理方法的优势。
- 13.26 说明异常不宜取代程序控制形式的原因。
- 13.27 说明处理相关异常的方法。

- 13.28 到本章为止,我们发现构造函数检测到的错误比较难以处理。异常处理则可以更好地处理这种错误。以 `String` 类的构造函数为例。这个构造函数用 `new` 取得自由空间。假如 `new` 操作失败,如何用非异常处理的其他方式来处理这种情况,讨论重点。指出如何使用异常来处理这种情况。并说明异常处理的优势。
- 13.29 假设程序抛出异常,开始执行相应的异常处理程序。再假设异常处理程序本身抛出相同的异常。这样会生成无穷递归吗?编写一个C++程序,验证结果是否正确。
- 13.30 用继承方法建立异常基类和各种派生类。然后显示指定基类的 `catch` 处理程序可捕捉派生类的异常。
- 13.31 列举返回 `double` 或 `int` 的条件异常。提供一个 `intcatch` 处理程序和一个 `double catch` 处理程序。说明不管返回 `double` 或 `int`,都只执行 `double catch` 处理程序。
- 13.32 编写一个C++程序,产生和处理内存溢出错误。程序通过操作符 `new` 循环请求生成动态存储空间。
- 13.33 编写一个C++程序,演示调用块中构造的对象的所有析构函数都在块中抛出异常之前得以调用。
- 13.34 编写一个C++程序,演示只调用发生异常之前构造的成员对象的成员对象析构函数。
- 13.35 编写一个C++程序,演示 `catch(...)` 如何捕捉异常。
- 13.36 编写一个C++程序,用以说明异常处理程序顺序的重要性。执行第一个匹配的异常处理程序。编译并运行程序,显示执行不同异常处理程序时的不同结果。
- 13.37 编写一个C++程序,表明构造函数向 `try` 块后的异常处理程序传递构造失败信息。
- 13.38 编写一个C++程序,用以说明使用异常类的多重继承层次时要考虑异常处理程序的顺序。
- 13.39 用 `setjmp/longjmp` 时,程序控制权可以立即从深层嵌套函数调用转移到错误程序。但由于堆栈解退,不能调用嵌套函数调用期间生成的自动对象的析构函数。编写一个C++程序,演示没有调用嵌套函数调用时生成的自动对象的析构函数。
- 13.40 编写一个C++程序,演示重抛出异常。
- 13.41 编写一个C++程序,用 `set _unexpected` 对 `unexpected` 设置用户自定义函数,再次用 `set _unexpected`,然后将 `unexpected` 恢复为原先的函数。编写一个类似的C++程序,测试 `set _terminate` 和 `terminate`。
- 13.42 编写一个C++程序,用以说明本身有 `try` 块的函数不必捕捉 `try` 块中产生的每个错误。有些错误可跳过,留给外层范围处理。
- 13.43 编写一个C++程序,令其从深层嵌套函数调用抛出错误,同时还包含 `catch` 处理程序,该程序后跟 `try` 块(用捕捉异常的调用链封闭起来)。

# 第 14 章 文件处理

## 学习目标

- 能建立、读、写和更新文件
- 熟悉顺序访问文件的处理方式
- 熟悉随机访问文件的处理方式
- 能指定高性能无格式的 I/O 操作
- 理解格式化与“原始数据”文件处理的区别
- 用随机访问文件处理方式编写事务处理程序

## 14.1 简介

变量和数组中保存的数据是临时的。文件用于永久保存大量数据。计算机把文件保存在二级存储设备中,如磁盘存储设备、光盘。本章要讨论怎样用C++ 程序建立、更新和处理数据文件,包括顺序存储文件和随机访问文件。还将比较格式化与“原始数据”文件处理的区别。第 19 章将介绍从字符串而非文件输入和输出数据。

## 14.2 数据的层次结构

计算机处理的所有数据项最终还原为是 0 和 1 的组合。这是因为制造表示两种稳定状态(分别用 0 和 1 表示)的电子设备既简单又经济。计算机要完成的复杂功能最后只涉及最基本的 0 和 1 操作。

0 和 1 可视为是计算机的最小数据项,人们称之为“位”,即 bit,它是二进制数(binarydigit)的缩写,一个二进制数是 0 和 1 的两个值之一。计算机电路完成各种简单的位操作,如确定某个位的值、设置某个位的值和反转某个位的值(0 变为 1,1 变为 0)等等。

让程序员来完成底层位的数据处理,势必非常麻烦。他们更喜欢用十进制数(即 0,1,2,3,4,5,6,7,8 和 9)、字母(即 A~Z 和 a~z)和专用符号(即 \$,@,%,&,\*,(,),- ,+ ,',:,?,/等等)来处理数据。数字、字母和专用符号称为“字符”(character)。特定计算机上用于编写程序和代表数据项的所有字符的集合称为“字符集”(character set)。因为计算机只能处理 1 和 0,所以计算机字符集中的每一个字符都是用名为字节(byte)的 0 和 1 组成的序列表示的。一个字节通常由 8 位构成。程序员以字符为单位创建程序和数据项,计算机按位序列操作和处理这些字符。

字符由位构成,域(field)由字符构成。域是一组有意义的字符。例如,一个只包含大写字母和小写字母的域可用于表示某人的名字。

计算机要处理的数据项组成了“数据层次结构”(data hierarchy)。数据层次结构中,数据项从位到字符再到域,越来越大、越来越复杂。

记录(如 C 语言中的 struct 或 class)由多个域构成(在 C++ 中称为成员)。例如,在一张工资表中,为某个特定雇员建立的一条记录可能由如下域组成:

- (1) 雇员标识号;
- (2) 名字;
- (3) 地址;
- (4) 每小时工资等级;
- (5) 免税申请号;
- (6) 年度收入;
- (7) 联邦税收额等等。

因此,记录是一组相关的域。上面的例子中,所有域对应于同一个雇员。当然,特定的公司会有许多雇员,所以要为每一个雇员建立一个工资表(记录)。文件是一组相关的记录。某个公司的工资表文件通常包含为每一个雇员建立的记录。较小公司的工资表文件可能只包含 22 条记录,而大型公司的工资表文件可能要包含 100 000 条记录。一个机构建立了成百上千个文件,而每个文件又可能包含几百万甚至几十亿个字符信息,这是很常见的事。图 14.1 表明了数据的层次结构。

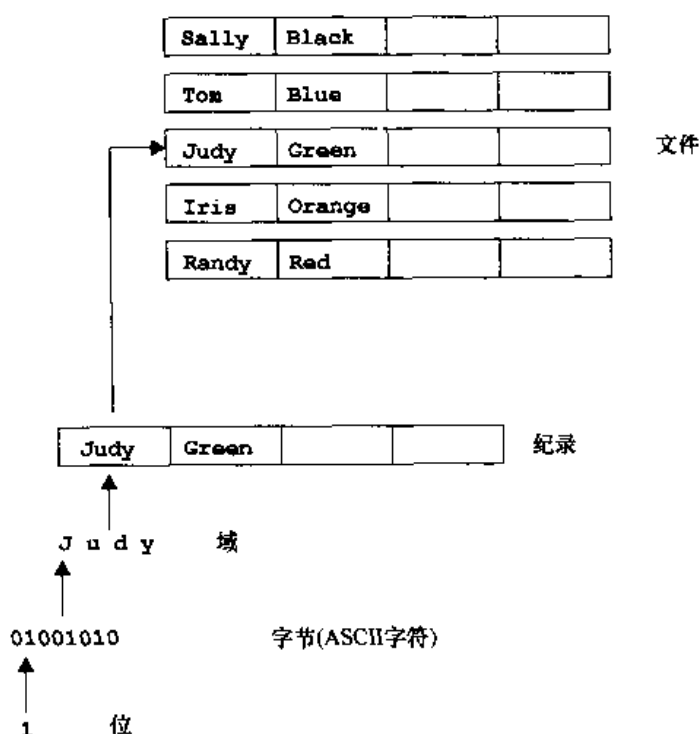


图 14.1 数据的层次结构

为了便于从文件获取特定的记录,每条记录中至少要选出一个域作为“记录关键字”(record key)。记录关键字标识了属于某人或某个实体而区别于文件中其他所有记录的记录。例如,在本节的工资表记录中,常把“雇员标识号”选作记录关键字。

组织文件中的记录有许多方式。最常见的组织方式是按记录关键字字段的顺序保存记

录,这种方式保存记录的文件称为“顺序文件”(sequential file)。在工资表文件中,记录通常按雇员标识号的顺序保存。在第一个雇员的记录中,该雇员的雇员标识号最小,其后的记录中包含的雇员标识号依次递增。

大多数的商业机构要用许多不同的文件来存储数据。例如,公司里可能要有工资表文件、应收账款目文件(列出客户的欠款)、应付账目文件(列出欠供应商的金额)、存货文件(列出经商的货物)和其他多种类型的文件。有时把一组相关的文件称为“数据库”(database)。为建立和管理数据库而设计的文件集合称为“数据库管理系统”(DBMS)。

### 14.3 文件和流

C++ 语言把每个文件都看成一个有序的字节流(如图 14.2 所示)。每一个文件不是以文件结束符(end-of-filemarker)结束,就是以在由系统维护和管理的数据结构中特定的字节号处结束。文件打开时,就会创建一个对象,将这个对象和某个流关联起来。第 11 章提到介绍过 cin,cout,cerr 和 clog 这 4 个对象会自动生成。与这些对象相关联的流提供程序与特定文件或设备之间的通信通道。例如,cin 对象(标准输入流对象)使程序能从键盘输入数据,cout 对象(标准输出流对象)使程序能向屏幕输出数据,cerr 和 clog 对象(标准错误流对象)使程序能向屏幕输出错误消息。



图 14.2 在 C++ 中查看  $n$  个字节的文件

在 C++ 中进行文件处理时,需要包括头文件 `<iostream.h>` 和 `<fstream.h>`。`<fstream.h>` 头文件包括流类 `ifstream`(从文件输入)、`ofstream`(向文件输出)和 `fstream`(从文件输入/输出)的定义。通过创建这些流类的对象来打开文件。这些流类分别从 `istream`, `ostream` 和 `iostream` 类派生(即继承它们的功能)。所以,第 11 章介绍的成员函数、操作符和流操纵元同样适用于文件流。I/O 类的继承关系如图 14.3 所示。

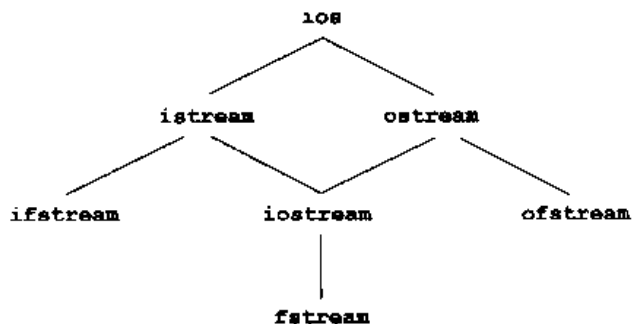


图 14.3 I/O 类的部分继承关系

## 14.4 创建顺序访问文件

C++ 把文件视为无结构的字节流,所以记录等说法在C++文件中是不存在的。为此,程序员必须将文件结构化以满足特定应用程序需求。下例演示了如何为文件强加记录结构。首先列出程序,然后再分析细节。

图 14.4 中的程序建立了一个简单的顺序访问文件,用于在应收账款管理系统中跟踪公司借贷客户的欠款数目。程序能够获取每一个客户的账号、客户名和客户的结算额(即客户以往收到货品和服务但未与公司结算的金额)。有关客户的数据就构成了该客户的记录。账号在应用程序中用作记录关键字,文件按账号顺序建立和维护。程序假定用户按账号顺序键入记录。在完善的应收账款管理系统中,应提供排序功能让用户按任意顺序键入记录,然后将排序后的记录写入文件。

```

1 //Fig.14.4: fig14_04.cpp
2 //Create a sequential file
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::ios;
8 using std::cerr;
9 using std::endl;
10
11 #include <fstream>
12
13 using std::ofstream;
14
15 #include <cstdlib>
16
17 int main()
18 |
19     //ofstream constructor opens file
20     ofstream outClientFile( "clients.dat", ios::out );
21
22     if ( ! outClientFile ) { //overloaded ! operator
23         cerr << "File could not be opened" << endl;
24         exit( 1 ); //prototype in cstdlib
25     }
26
27     cout << "Enter the account, name, and balance.\n"
28         << "Enter end - of - file to end input.\n? ";
29
30     int account;
31     char name[ 30 ];
32     double balance;
33
34     while ( cin >> account >> name >> balance ) |

```



```

35     outClientFile << account << ' ' << name
36             << ' ' << balance << '\n';
37     cout << "? ";
38     |
39
40     return 0; //ofstream destructor closes file
41 |

```

输出结果:

Enter the account, name, and balance.

Enter EOF to end input.

? 100 Jones 24.98

? 200 Doe 345.67

? 300 White 0.00

? 400 Stone -42.16

? 500 Rich 224.62

? ^z

图 14.4 创建顺序访问文件

接下来查看图 14.4 中的程序。如前所述,创建流类 `ifstream`、`ofstream` 或 `fstream` 对象后,便打开了文件。图 14.4 中,打开文件的目的是为了输出,所以创建了 `ofstream` 对象。文件名和文件打开方式两个参数被传给该对象的构造函数。对于 `ofstream` 对象,文件打开方式可以是 `ios::out`(将数据输出到文件)或 `ios::app`(将数据添加到文件末尾,不修改文件中现有的数据)。现有文件用 `ios::out` 打开时会被截掉,即文件中的所有数据均被删除。如果指定文件不存在,就用该文件名建立它。

第 20 行

```
ofstream outClientFile( "clients.dat", ios::out);
```

中的声明创建了一个 `ofstream` 对象 `outClientFile`,该对象与打开的、用于输出的文件 `clients.dat` 关联。参数“`clients.dat`”和 `ios::out` 被传入。打开了文件的 `ofstream` 构造函数。这样就和文件建立了通信。默认情况下,打开 `ofstream` 对象的目的是用于输出,因此下列语句

```
ofstream outClientFile( "clients.dat" );
```

也可以打开 `clients.dat` 进行输出。图 14.5 列出了文件打开方式。

| 文件打开方式                   | 描述                                           |
|--------------------------|----------------------------------------------|
| <code>ios::app</code>    | 将所有输出写入文件末尾                                  |
| <code>ios::ate</code>    | 打开文件以便输出,并移到文件末尾(通常用于在文件中添加数据)。数据可以写入文件的任何地方 |
| <code>ios::in</code>     | 打开文件以便输入                                     |
| <code>ios::out</code>    | 打开文件以便输出                                     |
| <code>ios::trunc</code>  | 删除文件中现有内容(这也是 <code>ios::out</code> 的默认操作)   |
| <code>ios::binary</code> | 打开文件以进行二进制(也就是非文本)格式输入或输出                    |

图 14.5 文件打开方式

**常见编程错误 14.1** 打开一个用户想保留数据的现有文件进行输出(以 `ios::out` 方式)。这种操作会在不给出任何警告消息的情况下删除文件内容。

### 常见编程错误 14.2 用错误的 ofstream 对象指定文件。

可以生成 ofstream 对象但不打开特定文件,可以在后期关联文件与对象。例如声明

```
ofstream outClientFile;
```

生成 ofstream 对象 outClientFile。ofstream 成员函数 open

```
outClientFile.open("clients.dat", ios::out );
```

打开文件并将其与现有 ofstream 对象关联。

### 常见编程错误 14.3 在引用文件之前忘记打开该文件。

创建 ofstream 对象并准备打开时,程序将测试打开操作是否成功。if 结构中的操作(第 22~25 行)

```
if ( ! outClientFile ) {
    cerr << "File could not be opened" << endl;
    exit( 1 );
}
```

用重载的 ios 操作符成员函数 operator! 确定打开操作是否成功。如果 open 操作的流已经设置了 failbit 或 badbit,该条件返回非 0 值(true)。可能的错误是试图打开读取不存在的文件、试图打开读取没有权限的文件或试图打开文件写入时却发现磁盘空间不足。

如果条件表示打开操作不成功,则输出错误消息“File could not be opened”,并调用函数 exit 中止程序,exit 的参数返回到调用该程序的环境中。参数 0 表示程序正常中止,其他任何值表示程序因某个错误而终止。exit 返回的值让调用环境(通常是操作系统)对错误做出相应响应。

另一个重载的 ios 操作符成员函数 operator void \* 将流变成指针,因此测试为 0(空指针)或非 0(任何其他指针值)。如果对流设置了 failbit 或 badbit(参见第 11 章),就会返回 0(false)。while 首部的条件

```
while ( cin >> account >> name >> balance )
```

自动调用 operator void \* 成员函数。只要不设置 cin 的 failbit 和 badbit,条件就会保持 true。输入文件结束符设置 cin 的 failbit。operator void \* 函数可以测试输入对象的文件结束符,无须再对输入对象显式调用 eof 成员函数。

如果文件打开成功,程序将开始处理数据。语句

```
cout << "Enter the account, name, and balance.\n"
    << " Enter EOF to and input.\n? ";
```

(第 27 行和第 28 行)提示用户对每个记录输入不同的域,或在数据输入完成时输入文件结束符。图 14.6 列出了不同计算机系统中文件结束符的键盘组合。

| 计算机系统       | 键盘组合                 |
|-------------|----------------------|
| UNIX 系统     | < Ctrl > d(在当前行)     |
| IBM PC 及兼容机 | < Ctrl > z(有时还需按回车键) |
| Macintosh   | < Ctrl > d           |
| VAX(VMS)    | < Ctrl > z           |

图 14.6 各种流行计算机系统中代表文件结束的键盘组合

## 第 34 行

```
while ( cin >> account >> name >> balance )
```

输入每组数据并确定是否输入了文件结束符。输入文件结束符或非法数据时, cin 的流读取操作符 >> 返回 0 (通常是流读取操作符返回, while 结构中中止。用户输入文件结束符告诉程序没有更多要处理的数据。当用户输入文件结束符组合键时, 设置文件结束符。只要不输入文件结束符, while 结构就会一直循环。

## 第 35 行和第 36 行

```
outClientFile << account << ' ' << name
               << ' ' << balance << '\n ';
```

程序开始处, 用流插入操作符 << 和程序开始处与文件关联的 outClientFile 对象, 将一组数据写入文件 clients.dat。可以用读取文件的程序取得这些数据 (参见 14.5 节)。注意图 14.4 中生成的文件是文本文件。可以用任何文本编辑器读取。

一旦输入文件结束符, main 就会中止, 这样会删除 outClientFile 对象, 从而调用其析构函数, 关闭文件 clients.dat。程序员可以用成员函数 close

```
outClientFile.Close();
```

显式关闭 ofstream 对象。

**性能提示 14.1** 程序不再引用的文件应立即显式关闭, 这样可以减少程序在不再需要特定文件之后继续执行所占用的资源。这种方法还可以使程序更清晰。

在图 14.4 所示的程序中, 用户输入了 5 条记录, 然后键入了表示数据输入结束的文件结束符 (IBM PC 兼容机的屏幕上显示 ^z)。输出结果的对话框中没有说明这些记录在文件中的组织形式。下一节将介绍读取和打印该文件的程序验证文件是否可用。

## 14.5 读取顺序访问文件中的数据

为了在需要时方便地检索要处理的数据, 数据应保存在文件中。前面讨论演示了如何建立一个顺序访问文件。这里要讨论按顺序读取文件中的数据。

图 14.7 中的程序读取文件 “clients.dat” (参图 14.4 中的程序) 中的记录, 并打印出了记录内容。通过建立 ifstream 类对象打开文件以便输入。向对象传入的两个参数是文件名和文件打开方式。声明

```
ifstream inClientFile(“ClientS.dat”, ios::in);
```

生成 ifstream 对象 inClientFile, 并将其与打开以便输入的文件 clients.dat 关联。括号中的参数传入 ifstream 构造函数, 打开文件并建立与文件的通道。

```
1 //Fig. 14.7: fig14_07.cpp
2 //Reading and printing a sequential file
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::ios;
8 using std::cerr;
9 using std::endl;
```

```

10
11 #include <fstream>
12
13 using std::ifstream;
14
15 #include <iomanip>
16
17 using std::setiosflags;
18 using std::resetiosflags;
19 using std::setw;
20 using std::setprecision;
21
22 #include <cstdlib>
23
24 void outputLine( int, const char * const, double );
25
26 int main()
27 {
28     //ifstream constructor opens the file
29     ifstream inClientFile( "clients.dat", ios::in );
30
31     if ( ! inClientFile ) {
32         cerr << "File could not be opened\n";
33         exit( 1 );
34     }
35
36     int account;
37     char name[ 30 ];
38     double balance;
39
40     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
41          << setw( 13 ) << "Name" << "Balance\n"
42          << setiosflags( ios::fixed | ios::showpoint );
43
44     while ( inClientFile >> account >> name >> balance )
45         outputLine( account, name, balance );
46
47     return 0; //ifstream destructor closes the file
48 }
49
50 void outputLine( int acct, const char * const name, double bal )
51 {
52     cout << setiosflags( ios::left ) << setw( 10 ) << acct
53          << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
54          << resetiosflags( ios::left )
55          << bal << '\n';
56 }

```

输出结果:

| Account | Name | Balance |
|---------|------|---------|
|---------|------|---------|

|     |       |        |
|-----|-------|--------|
| 100 | Jones | 24.98  |
| 200 | Doe   | 345.67 |
| 300 | White | 0.00   |
| 400 | Stone | -42.16 |
| 500 | Rich  | 224.62 |

图 14.7 读取和打印顺序文件

默认打开 `ifstream` 类对象进行输入,因此语句

```
ifstream inClientFile( "Clients.dat" );
```

可以打开 `clients.dat` 以便输入。和 `ofstream` 对象一样, `ifstream` 对象也可以生成而不打开特定文件,然后再将对象与文件关联。

**良好编程习惯 14.1** 如果文件内容不能修改,文件就只能打开用于输入(用 `ios::in`)。这可以避免不慎改动文件内容。这是最低访问权限原则的另一个例子。

程序用 `!inClientFile` 条件判断文件是否打开成功,然后再从文件中读取数据。语句

```
While( inClientFile >> account >> name >> balance )
```

从文件中读取一组值(即记录)。第一次执行完该条语句后, `account` 的值为 100, `name` 的值为 Jones, `balance` 的值为 24.98。每次执行程序中的该条语句时,函数都读取文件中的另一条记录,并把新的值赋给 `account`, `name` 和 `balance`。记录用函数 `outputLine` 显示,该函数用参数化流操纵元将数据格式化之后再显示。到达文件末尾时, `while` 结构中的输入序列返回 0 (通常返回 `inClientFile` 流), `ifstream` 析构函数将文件关闭,程序终止。

为了按顺序检索文件中的数据,程序通常要从文件起始位置开始读取数据,然后连续读取所有数据,直到找到所需要的数据为止。程序执行中可能需要按顺序从文件开始位置多次处理文件中的数据。 `ifstream` 类和 `ostream` 类都提供成员函数,使程序重新定位“文件位置指针”(读写操作所在的下一个字节号)。这些成员函数是 `ifstream` 类的 `seekg` (即 `seek get`) 和 `ostream` 类的 `seekp` (即 `seek put`)。每个 `istream` 对象有一个 `get` 指针,表示文件中下一个输入相同的字节数,每个 `ostream` 对象有一个 `put` 指针,表示文件中下一个输出相同的字节数。语句

```
inClientFile.seekg( 0 );
```

将文件位置指针移到文件开头(位置 0),连接 `inClientFile`。 `seekg` 的参数通常为 `long` 类型的整数。

第二个参数可以指定查找方向, `ios::beg` (默认)相对于流的开头定位, `ios::cur` 相对于流当前位置定位, `ios::end` 相对于流结尾定位。文件位置指针是个整数值,指定文件中离文件开头的相对位置(也称为距离文件开头的偏移量)。下面是一些 `get` 文件位置指针的例子:

```
//position to the nth byte of fileObject
//assumes ios::beg
fileObject.seekg( n );
```

```
//Position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
```

```
//position y bytes back from end of fileObject
FileObject.seekg( y, ios::end );

//position at end of fileObject
fileObject.seekg( 0, ios::end );
```

ostream 成员函数 seekp 也可以进行类似的操作。成员函数 tellg 和 tellp 分别返回 get 和 put 指针的当前位置。语句

```
Location = fileObject.tellg();
```

将 get 文件位置指针值赋给 long 类型的变量 location。通过图 14.8 中的程序,信用管理部经理可以了解这 3 类客户的情况:零余额账户(即客户不欠公司任何钱)、现金余额账户(即公司欠了其金额的客户)和负债余额账户(即因过去在该公司的购物或享受过服务而欠该公司一定金额的客户)。该程序显示了一个情单,允许信用管理部、经理输入了 3 个选项之一,获得信用信息。选项 1 显示了零余额账户清单,选项 2 显示现金余额账户情单。选项 3 显示负债余额账户情单。选项 4 终止程序执行。输入一个无效值会显示一条提示行,要求输入另外的选项。程序输出结果如图 14.9 所示。

```
1 //Fig. 14.8: fig14_CreditInquiry.cpp
2 //Credit inquiry program
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::ios;
8 using std::cerr;
9 using std::endl;
10
11 #include <fstream>
12
13 using std::ifstream;
14
15 #include <iomanip>
16
17 using std::setiosflags;
18 using std::resetiosflags;
19 using std::setw;
20 using std::setprecision;
21
22 #include <cstdlib>
23
24 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE,
25                  DEBIT_BALANCE, END };
26 int getRequest();
27 bool shouldDisplay( int, double );
28 void outputLine( int, const char * const, double );
29
30 int main()
31 {
32     //ifstream constructor opens the file
```

```
33     ifstream inClientFile( "clients.dat", ios::in );
34
35     if ( ! inClientFile ) {
36         cerr << "File could not be opened" << endl;
37         exit( 1 );
38     }
39
40     int request, account;
41     char name[ 30 ];
42     double balance;
43
44     cout << "Enter request \n"
45         << " 1 - List accounts with zero balances \n"
46         << " 2 - List accounts with credit balances \n"
47         << " 3 - List accounts with debit balances \n"
48         << " 4 - End of run"
49         << setiosflags( ios::fixed | ios::showpoint );
50     request = getRequest();
51
52     while ( request != END ) {
53
54         switch ( request ) {
55             case ZERO_BALANCE:
56                 cout << " \nAccounts with zero balances: \n";
57                 break;
58             case CREDIT_BALANCE:
59                 cout << " \nAccounts with credit balances: \n";
60                 break;
61             case DEBIT_BALANCE:
62                 cout << " \nAccounts with debit balances: \n";
63                 break;
64         }
65
66         inClientFile >> account >> name >> balance;
67
68         while ( ! inClientFile.eof() ) {
69             if ( shouldDisplay( request, balance ) )
70                 outputLine( account, name, balance );
71
72             inClientFile >> account >> name >> balance;
73         }
74
75         inClientFile.clear(); //reset eof for next input
76         inClientFile.seekg( 0 ); //move to beginning of file
77         request = getRequest();
78     }
79
80     cout << "End of run." << endl;
81
82     return 0; //ifstream destructor closes the file
83 }
```

```

84
85 int getRequest()
86 |
87     int request;
88
89     do |
90         cout << "\n? ";
91         cin >> request;
92     | while( request < ZERO_BALANCE && request > END );
93
94     return request;
95 |
96
97 bool shouldDisplay( int type, double balance )
98 |
99     if ( type == CREDIT_BALANCE && balance < 0 )
100         return true;
101
102     if ( type == DEBIT_BALANCE && balance > 0 )
103         return true;
104
105     if ( type == ZERO_BALANCE && balance == 0 )
106         return true;
107
108     return false;
109 |
110
111 void outputLine( int acct, const char * const name, double bal )
112 |
113     cout << setiosflags( ios::left ) << setw( 10 ) << acct
114             << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
115             << resetiosflags( ios::left )
116             << bal << '\n';
117 |

```

图 14.8 信用查询程序

输出结果:

Enter request

```

1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run
? 1

```

Accounts with zero balances:

```

300      White      0.00
? 2

```

Accounts with credit balances:

```

400      Stone     -42.16

```



```
Accounts with debit balances:
100      Jones      24.98
200      Doe        345.67
500      Rich       224.62

? 4
End of run.
```

图 14.9 图 14.8 所示程序的输出结果

## 14.6 更新顺序访问文件

14.4 节提到修改格式化和写入顺序访问文件的数据可能会破坏文件中其他数据。例如,如果需要把“White”改为“Worthington”,所得的结果并不是简单地改写原来的名字。White 的记录以如下形式

```
300 White 0.00
```

写入文件中。如果用新的名字从文件中相同的起始位置改写该记录,记录的格式就成为

```
300 Worthington 0.00
```

因为新记录的长度大于原始记录的长度,所以从“Worthington”的第二个“o”之后的字符将重定义文件中的下一条顺序记录。其原因在于:使用流插入操作符 << 和流读取操作符 >> 的格式化输入/输出模型中,域的宽度不定,所以记录的宽度也不定。例如,7,14,-117,2 047 和 27 383 都是 int 类型的值,虽然它们的内部存储占用相同的字节数,但是将它们以格式化文本打印到屏幕上或存储在磁盘上时占用的域则不一样。因此,格式化输入/输出模型通常不用于更新现有记录。

上述名字可以修改,但比较危险。比如,把 300 White 0.00 之前的记录复制到一个新文件中,然后写入新的记录并把 300 White 0.00 之后的记录复制到新文件中。这种方法要求在更新一条记录的同时,还要处理文件中的各条记录。如果文件中一次要更新许多记录,就可以使用这种方法。

## 14.7 随机访问文件

我们知道,生成顺序访问文件和从顺序访问文件搜索特定信息顺序访问文件不适宜快速访问应用程序,即立即找到特定记录的信息。快速访问应用程序的例子有航空订票系统、银行系统、销售网点系统、自动柜员机和其他要求快速处理特定数据的事务处理系统。银行面对的客户成千上万,但自动柜员机能在瞬间作出响应。这种快速访问应用程序是用随机访问文件实现的。随机访问文件的各个记录可以直接快速访问,而不需要进行搜索。

如前所述,C++ 不提供文件结构。因此应用程序要自己生成随机访问文件。虽然实现随机访问文件还有其它方法,但是本书的讨论只限于使用定长记录这种简洁明了的方法。因为随机访问文件中的每条记录都有相同的长度,所以能够用记录关键字的函数计算出每条记录相对于文件起始点的位置。后面将介绍如何立即访问文件甚至大型文件中指定的

记录。

图 14.10 反映了由定长记录组成的随机访问文件的一种观点(每条记录为 100 字节。可以将它比喻为一列火车,一些车箱是空的,一些车箱则满载货物,但是火车每节车箱的长度是相同。

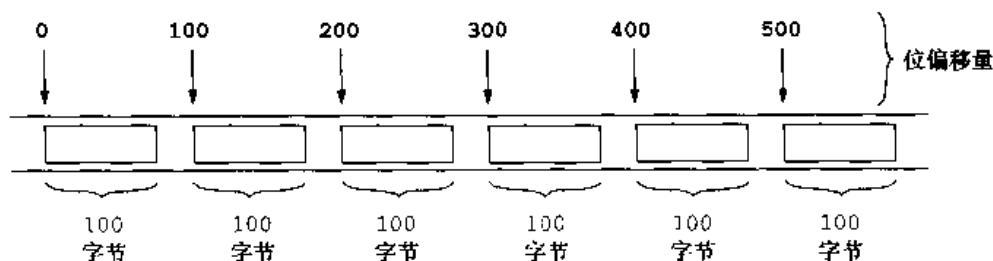


图 14.10 在 C++ 中查看随机访问文件

可以在不破坏其他数据的情况下把数据插入随机访问文件。也可在不改写整个文件的情况下更新和删除以前保存的数据。下面将讨论怎样建立随机访问文件、键入数据、按顺序和随机读取数据、更新数据和删除不再需要的数据。

## 14.8 建立随机访问文件

`ostream` 成员函数 `write` 把从内存中指定位置开始的固定个数的字节输出到指定流中。当流与文件关联时,数据被写到 `put` 文件位置指针所指的位置。`istream` 成员函数 `read` 把固定数目的字节从指定流输入到内存中指定地址开始的区域中。如果流与文件关联,则该字节从 `get` 文件位置指针指定的文件地址开始输入。

现在,将一个整型 `number` 写入文件时,对于 4 字节的整数,不再用语句

```
outFile << number;
```

打印 1 位或 11 位(10 位加一个符号位,各自需要 1 字节的存储空间),而改用语句

```
outFile.write( reinterpret_cast<const char*>(&number),
               sizeof( number ));
```

这种方法总是写入 4 字节(在 4 字节整数机器上)。`write` 函数要求一个 `const char *` 类型的数据作为它的第一个参数,因此我们用 `reinterpret_cast<const char*>` 强制类型转换操作符将 `number` 的地址变为 `const char *` 指针。`write` 的第二个参数是 `size_t` 类型的整数,指定写入的字节数。可以看出,`istream` 函数 `read` 可以将 4 个字节读回整型变量 `number` 中。

如果程序要读取用 `write` 写入的无格式数据,它必须在与执行写数据兼容的系统上编译和执行。

随机存取文件处理程序很少在文件中只写入一个域。通常会一次写入一个 `struct` 或一个 `class` 对象,如下例子所示。

分析以下问题陈述:

建立一个能够存储 100 个定长记录的借贷处理程序,供拥有 100 个以上客户的一家公司使用。每一条记录应包括账号(作为记录关键字)、姓、名和借贷金额。程序要能够更新、

插入和删除一条记录以及能够以格式化文本形式列出所有记录。

下面将介绍建立该借贷处理程序所需的技术。图 14.11 中的程序说明了怎样打开一个随机访问文件、怎样用 struct 定义一条记录格式(定义见在 clntdata.h 头文件)以及怎样以二进制形式把数据写入磁盘(第 33 行指定二进制形式)。程序用 write 函数和空结构初始化了文件 credit.dat 的所有 100 条记录。每一个空结构中,账号都为 0,姓氏和名为 NULL,借贷金额为 0.00,以这种方式初始化后就在磁盘上建立了存储文件的空间,并且能够在后面的程序中确定某条记录是否包含数据。

```

1 //Fig. 14.11: clntdata.h
2 //Definition of struct clientData used in
3 //Figs. 14.11, 14.12, 14.14 and 14.15.
4 #ifndef CLNTDATA_H
5 #define CLNTDATA_H
6
7 struct clientData {
8     int accountNumber;
9     char lastName[ 15 ];
10    char firstName[ 10 ];
11    double balance;
12 };
13
14 #endif

```

图 14.11 头文件 clntdata.h

```

15 //Fig. 14.11: fig14_11.cpp
16 //Creating a randomly accessed file sequentially
17 #include <iostream>
18
19 using std::cerr;
20 using std::endl;
21 using std::ios;
22
23 #include <fstream>
24
25 using std::ofstream;
26
27 #include <cstdlib>
28
29 #include "clntdata.h"
30
31 int main()
32 {
33     ofstream outCredit( "credit.dat", ios::binary );
34
35     if ( ! outCredit ) {
36         cerr << "File could not be opened." << endl;
37         exit( 1 );
38     }

```

```

39
40   clientData blankClient = { 0, "", "", 0.0 };
41
42   for ( int i = 0; i < 100; i++ )
43       outCredit.write(
44           reinterpret_cast<const char*>( &blankClient ),
45           sizeof( clientData ) );
46   return 0;
47 }
```

图 14.11 按顺序创建随机访问文件

在图 14.11 中,语句(第 43~45 行)

```

outCredit.write(
    reinterpret_cast<const char*>( &blankClient ),
    sizeof( ClientData ) );
```

将长度为 `sizeof( clientData )` 的 `blankClient` 结构写入与 `ofstream` 的对象 `outCredit` 关联的文件 `credit.dat`。记住,操作符 `sizeof` 返回括号中对象的长度(详情参见第 5 章)。注意,第 43 行函数 `write` 的第一个参数应为 `const char*` 类型,但 `&blankClient` 的数据类型为 `clientData*`。要将 `&blankClient` 变为相应指针类型,表达式

```
reinterpret_cast<const char*>(&blankClient)
```

用强制类型转换操作符 `reinterpret_cast` 将 `blankClient` 地址转换为 `const char*` 类型,因此调用 `write` 可以实现顺利编译,不会产生语法错误。

## 14.9 向随机访问文件随机写入数据

图 14.12 中的程序把数据写入文件 `credit.dat` 中。`ostream` 的函数 `seekp` 和 `write` 用于将数据保存储文件中指定的位置。程序先用函数 `seekp` 把 `put` 文件位置指针指向文件中指定的位置,然后用 `write` 函数写入数据。输出结果如图 14.13 所示。注意图 14.12 中的程序(第 16 行)包括了图 14.11 中定义的头文件 `clntdata.h`。

```

1 //Fig.14.12: fig14_12.cpp
2 //Writing to a random access file
3 #include <iostream>
4
5 using std::cerr;
6 using std::endl;
7 using std::cout;
8 using std::cin;
9 using std::ios;
10
11 #include <fstream>
12
13 using std::ofstream;
14
15 #include <cstdlib>
```

```
16 #include "clntdata.h"
17
18 int main()
19 |
20     ofstream outCredit( "credit.dat", ios::binary );
21
22     if ( ! outCredit ) {
23         cerr << "File could not be opened." << endl;
24         exit( 1 );
25     }
26
27     cout << "Enter account number "
28         << "(1 to 100, 0 to end input) \n? ";
29
30     clientData client;
31     cin >> client.accountNumber;
32
33     while ( client.accountNumber > 0 &&
34         client.accountNumber <= 100 ) {
35         cout << "Enter lastname, firstname, balance \n? ";
36         cin >> client.lastName >> client.firstName
37             >> client.balance;
38
39         outCredit.seekp( ( client.accountNumber - 1 ) *
40             sizeof( clientData ) );
41         outCredit.write(
42             reinterpret_cast<const char * >( &client ),
43             sizeof( clientData ) );
44
45         cout << "Enter account number \n? ";
46         cin >> client.accountNumber;
47     }
48
49     return 0;
50 |
```

图 14.12 按顺序创建随机访问文件

输出结果:

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
```

```

Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0

```

图 14.13 图 14.12 所示程序的输出结果

### 第 39 行和第 40 行

```

outCredit.seekp( ( client.accountNumber - 1 ) *
                SizeOf( ClientData ) );

```

将 outCredit 对象的“put”文件位置指针放在  $(\text{client.accountNumber} - 1) * \text{sizeof}(\text{clientData})$  求出的字节位置处。由于账号在 1 到 100 之间,因此计算记录的字节位置时要从账号减 1。这样,对于记录 1,文件位置指针设置为文件的字节 0。注意 ofstream 的对象 outCredit 用文件打开方式 ios::ate 打开。“put”文件位置指针最初设置为文件末尾,但数据可以在文件任何位置写入。

## 14.10 从随机访问文件中顺序读取数据

前面创建了随机访问文件并将数据写入该文件。这里要开发一个程序,顺序读取这个文件,只打印包含数据的记录。该程序还有另一好处,详情参见本节最后。

istream 的函数 read 从指定流的当前位置向对象输入指定字节数。例如,图 14.14 中的语句

```

inCredit.read( reinterpret_cast<char*>( &client ),
              sizeof( clientData ) );

```

从与 ifstream 的对象 inCredit 关联的文件中读取  $\text{sizeof}(\text{clientData})$  指定的字节数,并将数据保存在结构 client 中。注意函数 read 要求第一个参数类型为 char\*。由于 &client 的类型为 clientData,因此 &client 要用强制类型转换操作符 reinterpret\_cast 变为 char\*。注意图 14.14 中的程序包括图 14.11 定义的头文件 clntdata.h。

```

1 //Fig.14.14; fig14_14.cpp
2 //Reading a random access file sequentially
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8 using std::cerr;
9
10 #include <iomanip>
11
12 using std::setprecision;

```

```
13 using std::setiosflags;
14 using std::resetiosflags;
15 using std::setw;
16
17 #include <fstream>
18
19 using std::ifstream;
20 using std::ostream;
21
22 #include <cstdlib>
23 #include "clntdata.h"
24
25 void outputLine( ostream&, const clientData & );
26
27 int main()
28 {
29     ifstream inCredit( "credit.dat", ios::in );
30
31     if ( ! inCredit ) {
32         cerr << "File could not be opened." << endl;
33         exit( 1 );
34     }
35
36     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
37          << setw( 16 ) << "Last Name" << setw( 11 )
38          << "First Name" << resetiosflags( ios::left )
39          << setw( 10 ) << "Balance" << endl;
40
41     clientData client;
42
43     inCredit.read( reinterpret_cast<char*>( &client ),
44                  sizeof( clientData ) );
45
46     while( inCredit && ! inCredit.eof() ) {
47
48         if ( client.accountNumber != 0 )
49             outputLine( cout, client );
50
51         inCredit.read( reinterpret_cast<char*>( &client ),
52                      sizeof( clientData ) );
53     }
54
55     return 0;
56 }
57
58 void outputLine( ostream& output, const clientData &c )
59 {
60     output << setiosflags( ios::left ) << setw( 10 )
61          << c.accountNumber << setw( 16 ) << c.lastName
62          << setw( 11 ) << c.firstName << setw( 10 )
63          << setprecision( 2 ) << resetiosflags( ios::left )
```

```

64         << setiosflags( ios::fixed | ios::showpoint )
65         << c.balance << '\n';
66     }

```

输出结果:

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

图 14.14 按顺序读取随机访问文件

图 14.14 中的程序顺序读取 credit.dat 文件中的每条记录,检查每个记录中是否包含数据,并打印包含数据的记录。第 46 行

```
while (inCredit && ! inCredit.eof()) {
```

用 ios 成员函数 eof 确定是否到达文件末尾,到达文件末尾,就终止执行 while 结构。如果读取文件时发生错误,循环终止,因为 inCredit 值为 false。从文件中输入的数据用 outputLine 输出,outputLine 取两个参数,即 ostream 对象和要输出的 clientData 结构。ostream 参数类型可以支持任何 ostream 对象(如 cout)或 ostream 的任何派生类对象(如 ofstream 类型的对象)作为参数。这样,同一函数既可输出到标准输出流,也可输出到文件流,而不必编写另一个函数。

这些程序还有优势吗?看看输出窗口,你会发现记录已经按账号排序列出,这是用直接访问方法将这些记录存放到文件中的结果。试比较第 4 章介绍的冒泡排序,用直接访问方法排序显然快得多。这是因为生成的文件足够大可保证完成每条记录。当然,大多数时候文件存储都不紧凑,所以会浪费存储空间。这是以牺牲空间换取效率的另一个示例:加大空间可获得更快的排序算法。

## 14.11 案例分析:事务处理程序

下面介绍一个有实际意义的使用随机访问文件的事务处理程序(如图 14.15 所示)。该程序维护银行的账号信息。程序能够更新、添加和删除账号,并且能够把所有当前账号的格式化清单保存在一个用于打印的文本文件中。我们假定已经通过执行图 14.11 中的程序建立了文件 credit.dat,并用图 14.12 中的程序插入初始值。

```

1  //Fig.14.15: fig14_15.cpp
2  //This program reads a random access file sequentially,
3  //updates data already written to the file, creates new
4  //data to be placed in the file, and deletes data
5  //already in the file.
6  #include <iostream>
7
8  using std::cout;
9  using std::cerr;

```



```
10  using std::cin;
11  using std::endl;
12  using std::ios;
13
14  #include <fstream>
15
16  using std::ofstream;
17  using std::ostream;
18  using std::fstream;
19
20  #include <iomanip>
21
22  using std::setiosflags;
23  using std::resetiosflags;
24  using std::setw;
25  using std::setprecision;
26
27  #include <cstdlib>
28  #include "clntdata.h"
29
30  int enterChoice();
31  void textFile( fstream& );
32  void updateRecord( fstream& );
33  void newRecord( fstream& );
34  void deleteRecord( fstream& );
35  void outputLine( ostream&, const clientData & );
36  int getAccount( const char * const );
37
38  enum Choices { TEXTFILE = 1, UPDATE, NEW, DELETE, END };
39
40  int main()
41  {
42      fstream inOutCredit( "credit.dat", ios::in | ios::out );
43
44      if ( ! inOutCredit ) {
45          cerr << "File could not be opened." << endl;
46          exit ( 1 );
47      }
48
49      int choice;
50
51      while ( ( choice = enterChoice() ) != END ) {
52
53          switch ( choice ) {
54              case TEXTFILE:
55                  textFile( inOutCredit );
56                  break;
57              case UPDATE:
58                  updateRecord( inOutCredit );
59                  break;
60              case NEW:
```

```

61         newRecord( inOutCredit );
62         break;
63     case DELETE:
64         deleteRecord( inOutCredit );
65         break;
66     default:
67         cerr << "Incorrect choice\n";
68         break;
69     }
70
71     inOutCredit.clear(); //resets end-of-file indicator
72 }
73
74 return 0;
75 }
76
77 //Prompt for and input menu choice
78 int enterChoice()
79 {
80     cout << "\nEnter your choice" << endl
81         << "1 - store a formatted text file of accounts\n"
82         << "    called 'print.txt' for printing\n"
83         << "2 - update an account\n"
84         << "3 - add a new account\n"
85         << "4 - delete an account\n"
86         << "5 - end program\n? ";
87
88     int menuChoice;
89     cin >> menuChoice;
90     return menuChoice;
91 }
92
93 //Create formatted text file for printing
94 void textFile( fstream &readFromFile )
95 {
96     ofstream outPrintFile( "print.txt", ios::out );
97
98     if ( ! outPrintFile ) {
99         cerr << "File could not be opened." << endl;
100         exit( 1 );
101     }
102
103     outPrintFile << setiosflags( ios::left ) << setw( 10 )
104         << "Account" << setw( 16 ) << "Last Name" << setw( 11 )
105         << "First Name" << resetiosflags( ios::left )
106         << setw( 10 ) << "Balance" << endl;
107     readFromFile.seekg( 0 );
108
109     clientData client;
110     readFromFile.read( reinterpret_cast<char*>( &client ),
111         sizeof( clientData ) );

```

```
112
113     while ( ! readFromFile.eof() ) {
114         if ( client.accountNumber != 0 )
115             outputLine( outPrintFile, client );
116
117         readFromFile.read( reinterpret_cast<char * >( &client ),
118                             sizeof( clientData ) );
119     }
120 }
121
122 //Update an account's balance
123 void updateRecord( fstream &updateFile )
124 {
125     int account = getAccount( "Enter account to update" );
126
127     updateFile.seekg( ( account - 1 ) * sizeof( clientData ) );
128
129     clientData client;
130     updateFile.read( reinterpret_cast<char * >( &client ),
131                     sizeof( clientData ) );
132
133     if ( client.accountNumber != 0 ) {
134         outputLine( cout, client );
135         cout << "\nEnter charge ( + ) or payment ( - ): ";
136
137         double transaction; //charge or payment
138         cin >> transaction; //should validate
139         client.balance += transaction;
140         outputLine( cout, client );
141         updateFile.seekp( ( account - 1 ) * sizeof( clientData ) );
142         updateFile.write(
143             reinterpret_cast<const char * >( &client ),
144             sizeof( clientData ) );
145     }
146     else
147         cerr << "Account #" << account
148              << " has no information." << endl;
149 }
150
151 //Create and insert new record
152 void newRecord( fstream &insertInFile )
153 {
154     int account = getAccount( "Enter new account number" );
155
156     insertInFile.seekg( ( account - 1 ) * sizeof( clientData ) );
157
158     clientData client;
159     insertInFile.read( reinterpret_cast<char * >( &client ),
160                       sizeof( clientData ) );
161
162     if ( client.accountNumber == 0 ) {
```

```

163     cout << "Enter lastname, firstname, balance\n? ";
164     cin >> client.lastName >> client.firstName
165         >> client.balance;
166     client.accountNumber = account;
167     insertInFile.seekp( ( account - 1 ) *
168                         sizeof( clientData ) );
169     insertInFile.write(
170         reinterpret_cast<const char * >( &client ),
171         sizeof( clientData ) );
172 }
173 else
174     cerr << "Account #" << account
175         << " already contains information." << endl;
176 }
177
178 //Delete an existing record
179 void deleteRecord( fstream &deleteFromFile )
180 {
181     int account = getAccount( "Enter account to delete" );
182
183     deleteFromFile.seekg( (account - 1) * sizeof( clientData ) );
184
185     clientData client;
186     deleteFromFile.read( reinterpret_cast<char * >( &client ),
187                         sizeof( clientData ) );
188
189     if ( client.accountNumber != 0 ) {
190         clientData blankClient = { 0, "", "", 0.0 };
191
192         deleteFromFile.seekp( ( account - 1 ) *
193                               sizeof( clientData ) );
194         deleteFromFile.write(
195             reinterpret_cast<const char * >( &blankClient ),
196             sizeof( clientData ) );
197         cout << "Account #" << account << " deleted." << endl;
198     }
199     else
200         cerr << "Account #" << account << " is empty." << endl;
201 }
202
203 //Output a line of client information
204 void outputLine( ostream &output, const clientData &c )
205 {
206     output << setiosflags( ios::left ) << setw( 10 )
207         << c.accountNumber << setw( 16 ) << c.lastName
208         << setw( 11 ) << c.firstName << setw( 10 )
209         << setprecision( 2 ) << resetiosflags( ios::left )
210         << setiosflags( ios::fixed | ios::showpoint )
211         << c.balance << '\n';
212 }
213

```

```

214 //get an account number from the keyboard
215 int getAccount( const char * const prompt )
216 {
217     int account;
218
219     do {
220         cout << prompt << " (1 - 100): ";
221         cin >> account;
222     } while ( account < 1 || account > 100 );
223
224     return account;
225 }

```

图 14.15 银行账号程序

程序有 5 个选项(第 5 个选项用于终止程序)。选项 1 调用函数 `textFile` 把所有格式化的账号保存在文本文件 `print.txt` 中(以后可能要打印这个文件)。函数 `textFile` 取一个 `fstream` 对象作为参数,用于从 `credit.dat` 文件输入数据。函数 `textFile` 用 `istream` 成员函数 `read` 和图 14.14 介绍的顺序文件访问方法从 `credit.dat` 输入数据。使用 14.10 节讨论的函数 `outputLine` 将数据输出到 `print.txt` 文件。注意 `textFile` 用 `istream` 成员函数 `seekg` 保证文件位置指针位于文件开头。选择了选项 1 后,文件 `accounts.txt` 中包含如下内容

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

选项 2 调用函数 `updateRecord` 更新账号。该函数只更新已有的记录,所以函数首先检查用户指定的记录是否为空。用 `istream` 成员函数 `read` 把记录读到结构 `client` 中,然后把比较成员 `client.accountNumber` 与 0。如果 `client.accountNumber` 为 0,说明该条记录中不包含信息,所以打印说明该记录为空的消息,然后再显示选项菜单。如果记录中包含信息,函数 `updateRecord` 用函数 `outputLine` 在屏幕上显示记录,并输入事务金额、计算新的结算结果以及把记录重新写文件。选项 2 的典型输出为

```

Enter account to update (1 -100): 37
37      Barker      Doug      0.00
Enter Charge ( + ) or payment ( - ): +87.99
37      Barker      Doug      87.99

```

选项 3 调用函数 `newRecord` 把新的账号添入文件。如果用户键入了一个已有账号,函数 `newRecord` 会指出该账号已存在,并再次显示出选项菜单。函数添加新记录的过程与图 14.12 中的程序所用的方法相同。选项 3 的典型输出为

```

Enter new account number(1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45

```

选项 4 调用函数 `deleteRecord` 删除文件中的一条记录。提示用户输入账号,只能删除有记录,如果该账号的记录为空,函数将显示账号不存在的错误消息。如果该账号已存在,通过将空记录(`blankClient`)复制到文件中重新初始化该记录。删除记录时会显示一条消息。选项 4 的典型输出如下所示

```
Enter account to delete ( 1-100 ): 29
Account #29 deleted.
```

打开 `credit.dat` 文件时,要结合使用 `ios::in` 和 `ios::out` 方式生成 `fstream` 对象,以便读写。

## 14.12 对象的输入/输出

本章和第 11 章介绍了 C++ 的面向对象的输入/输出。但我们的例子重点在于传统数据类型的 I/O 而非用户自定义类对象的 I/O。第 8 章介绍了如何用操作符重载输入与输出类对象。我们通过相应的 `istream` 重载流读取操作符 `>>` 进行对象输入,通过相应的 `ostream` 重载流插入操作符 `<<` 进行对象输出。两种情况下都只输入和输出对象的数据成员,而且都以对特定的抽象数据类型对象面言有意义的方式进行。对象成员函数在计算机内部提供,在数据输入时通过重载流插入操作符而与数据值组合。

对象的数据成员输出到磁盘文件时,会丢失对象的类型信息。保存盘的只有数据,没有类型信息。如果读取这个数据的程序知道其对应的对象类型,数据就能读取该类型的对象。

同一文件中如果保存不同类型的对象,会发生有趣的问题,如何在将其读入程序中时区分它们(或其数据成员集合)呢?当然,问题在于对象通常没有类型域(详情参见第 10 章)。

方法之一是让每个重载的输出操作符输出类型代码,放在表示一个对象的数据成员集合之前。然后对象输入始终以读取类型代码域开头,并用 `switch` 语句调用相应的重载函数。尽管这个方法不如多态编程巧妙,但它提供的机制,可在文件中保持对象并在需要进行读取。

## 14.13 小结

- 计算机处理的所有数据项最终还原为 0 和 1 的组合。
- 0 和 1 可视为计算机中的最小数据项是,该数据项称为“位”。
- 数字、字母和专用符号称为“字符”。能够在特定计算机上用于编写程序和代表数据项的所有字符的集合称为“字符集”。因为计算机只能处理 1 和 0,所以计算机字符集中的每个字符都是用名为字节(byte)的 0 和 1 序列表示的。
- 域是一组有意义的字符。
- 记录是一组相关的域。
- 每条记录中通常至少要选出一个域作为“记录关键字”。记录关键字标识了文件中属于某人或某个实体的记录。
- 在文件中组织记录最常用的方法是把记录组织成顺序访问文件。
- 为建立和管理数据库面设计的程序集合称为“数据库管理系统”(DBMS)。

- C++ 语言把每个文件都看作一个有序的字节流。
- 每一个文件根据与机器相关的文件结束符结束。
- 流提供文件与程序之间的通信通道。
- 要在C++ 中进行文件的 I/O 处理,就要包括头文件 `<iostream.h>` 和 `<fstream.h>`。  
`<fstream.h>` 首部包括流类 `ifstream`, `ofstream` 和 `fstream` 的定义。
- 通过建立 `ifstream`, `ofstream` 或 `fstream` 流类对象而打开文件。
- 因为C++ 把文件视为无结构的字节流,所以记录等说法在C++ 语言中并不存在。为此,程序员必须提供满足特定应用程序要求的文件结构。
- 通过生成 `ofstream` 对象打开文件以便输出。向对象传递两个参数——文件名和文件打开方式。
- 对于 `ofstream` 对象,文件打开方式可取 `ios::out` (将数据输出到文件)或 `ios::app` (将数据添加到文件末尾,而不修改文件中现有的数据)。现有文件用 `ios::out` 打开时会截尾,即文件中的所有数据均删除。如果指定文件还不存在,就用该文件名新建这个文件。
- 用 `ios` 操作符成员函数 `operator!` 确定打开操作是否成功。如果 `open` 操作的流将 `failbit` 或 `badbit` 设置,则这个条件返回非 0 值(`true`)。
- 程序可以不处理文件、处理一个文件或处理几个文件。每个文件有惟一的名称,与相应的文件流对象相关联。所有文件处理函数还引用相应对象的文件。
- `istream` 类和 `ostream` 类都提供成员函数,使程序把“文件位置指针”重新定位。这些成员函数是 `istream` 类的 `seekg`(`seekget`)和 `ostream` 类的 `seekp`(`seekput`)。每个 `istream` 对象有一个 `get` 指针,表示文件中下一个输入相距的字节数;每个 `ostream` 对象有一个 `put` 指针,表示文件中下一个输出相距的字节数。
- 成员函数 `tellg` 和 `tellp` 分别返回 `get` 和 `put` 指针的当前位置。
- 实现随机访问文件的简便方法是只用定长记录。这样,程序就可以迅速计算记录相对于文件开头的具体位置。
- 可以在不破坏其他数据的情况下把数据插入到随机访问文件。也能在不重写整个文件的情况下更新和删除以前存储的数据。
- `ostream` 成员函数 `write` 把从内存中指定位置开始的固定个数的字节送到指定流中,当流与文件关联时,数据写入到 `put` 文件位置指针所指示的位置。
- `istream` 成员函数 `read` 把一定的字节数从指定流输入到内存中指定地址开始的区域。该字节从 `get` 文件位置指针指定的文件地址开始输入。
- `write` 函数要求一个 `const char *` 类型的参数为第一个参数,因此我们用强制类型转换操作符将其他类型的地址变为 `const char *` 指针。
- 编译时,一元操作符 `sizeof` 返回括号中对象的长度(字节数),`sizeof` 返回无符号整数。
- `istream` 函数 `read` 从指定流的当前位置向对象输入指定字节数,`read` 要求第一个参数类型为 `char *`。
- `ios` 成员函数 `eof` 用于判断是否到达文件末尾,如果读取文件发生错误,就要设置文件结束符。

## 本章术语

|                                  |                                                |
|----------------------------------|------------------------------------------------|
| alphabetic field 字母域             | in - core I/O 内核 I/O                           |
| binary digit 二进制数                | in - memory I/O 内存 I/O                         |
| bit 位                            | input stream 输入流                               |
| byte 字节                          | istream class istream 类                        |
| cerr(standard error unbuffered)  | numericfield 数字域                               |
| cerr(无缓冲标准错误流)                   | open a file 打开文件                               |
| character field 字符域              | open member function open 成员函数                 |
| character set 字符集                | operator void * member function                |
| cin(standard input) cin(标准输入)    | operator void * 成员函数                           |
| clog(standard error buffered)    | operator! memberfunction                       |
| clog(缓冲标准错误流)                    | operator! 成员函数                                 |
| close a file 关闭文件                | ostream class ostream 类                        |
| close memberfunction close 成员函数  | ostream class ostream 类                        |
| cout(standard output) cout(标准输出) | output stream 输出流                              |
| data hierarchy 数据层次结构            | random accessfile 随机访问文件                       |
| database management system(DBMS) | record key 记录关键字                               |
| 数据库管理系统(DBMS)                    | record 记录                                      |
| database 数据库                     | seekg istream member function                  |
| decimaldigit 十进制数                | seekg istream 成员函数                             |
| ends streammanipulator ends 流操纵元 | seekp ostream member function                  |
| field 域                          | seekp ostream 成员函数                             |
| file position pointer 文件位置指针     | sequential accessfile 顺序访问文件                   |
| filename 文件名                     | special symbol 特殊符号                            |
| file 文件                          | stream 流                                       |
| fstream classfstream 类           | tellgistream member function tellgistream 成员函数 |
| fstream.hheaderfilefstream 头文件   | tellpostream member function tellpostream 成员函数 |
| ifstreamclass ifstream 类         | truncate an existing file 截尾现有文件               |

## 常见编程错误

- 14.1 打开一个用户想保留数据的现有文件进行输出(以 ios::out 方式)。这种操作会在不给出任何警告消息的情况下删除文件内容。
- 14.2 用错误的 ofstream 对象指定文件。
- 14.3 在引用文件之前忘记打开该文件。

## 良好编程习惯

- 14.1 如果文件内容不能修改,就只能打开文件输入(用 ios::in),以避免不慎改动文件。这是最低权限原则的又一个示例。

## 性能提示

- 14.1 程序不再引用的文件应立即显式关闭,这样可以减少程序在不再需要特定文件之后继续执行所占用的资源。这种方法还可以使程序更清晰。



## 自测题

## 14.1 填空题:

- a) 计算机处理的所有数据项最终都是\_\_\_\_\_和\_\_\_\_\_的组合。
- b) 计算机所能处理的最小数据项称为\_\_\_\_\_。
- c) \_\_\_\_\_是一组相关的记录。
- d) 数字、字母和专用符号称为\_\_\_\_\_。
- e) 一组相关的文件称为\_\_\_\_\_。
- f) 类 ofstream、ifstream 和 ostream 文件流的成员函数\_\_\_\_\_关闭文件。
- g) istream 成员函数\_\_\_\_\_从指定流中读取一个字符。
- h) istream 成员函数\_\_\_\_\_和\_\_\_\_\_从指定流中读取一行数据。
- i) Ofstream, ifstream 和 ostream 文件流类成员函数\_\_\_\_\_打开一个文件。
- j) 以随机访问方式读取文件中的数据通常使用 istream 成员函数\_\_\_\_\_。
- k) istream 和 ostream 类成员函数\_\_\_\_\_、\_\_\_\_\_把文件位置指针重定位到输入流与输出流中指定的位置。

## 14.2 判断正误。如果有错,请说明原因。

- a) 函数 read 不能从标准输入流对象 cin 读取数据。
- b) 程序员必须显式生成 cin, cout, cerr 和 clog 对象。
- c) 程序必须明确调用函数 close 关闭与 fstream, ifstream 和 ofstream 对象关联的文件。
- d) 如果文件位置指针没有指向顺序访问文件的起始位置,要从文件起始位置读取数据,必须先关闭文件然后再打开它。
- e) ostream 成员函数 write 能够把数据写入标准输出流 cout。
- o) 更新顺序访问文件中的数据一般不会改写其他数据。
- g) 查找随机访问文件中的指定记录不必从头逐条查找。
- h) 随机访问文件中的记录必须有统一的长度。
- i) 函数 seekp 和 seekg 只能定位相对于文件起始点的位置。

## 14.3 假定下列每一条语句用于同一个程序。

- a) 编写一条语句,打开文件 oldmast. dat 以便输入数据,用 ifstream 对象 inOldMaster。
- b) 编写一条语句,打开文件 trans. dat 以便输入数据,使用 ifstream 对象 Transaction。
- c) 编写一条语句,打开文件 newmast. dat 以便输出(以及建立)数据,使用 ofstream 对象 outNewMaster。
- d) 编写一条语句,读取文件 oldmast. dat 中的记录。记录由整数 accountNum、字符串 name 和浮点数 currentBalance 组成,使用 ifstream 对象 inOldMaster。
- e) 编写一条语句,读取文件 trans. dat 中的一条记录。记录由整数 accountNum 和浮点数 dollarAmount 组成,使用 ifstream 对象 inTransaction。
- f) 编写一条语句,向文件 newmast. dat 中写入一条记录,记录由整数 accountNum、字符串 name 和浮点数 currentBalance 组成,使用 ofstream 对象 outNewMaster。

## 14.4 指出下列程序中的错误,并说明如何改正。

- a) ofstream 对象 outPayable 所引用的文件(payables. dat)尚未打开。

```
outPayable << account << company << amount << endl;
```

- b) 下一条语句要从文件 payables. dm 中读取一条记录。ifstream 对象 inPayable 引用了

该文件,istream 对象 inReceivable 引用了文件 receivables.dat。

```
inReceivable >> account >> company >> amount;
```

c) 打开文件 tools.dat,在不删除当前数据的情况下把数据添加到文件中。

```
ofstream outTools("tools.dat", ios::out);
```

### 自测题答案

14.1 a) 1, 0 b) 位 c) 文件 d) 字符 e) 数据库 f) close g) get h) get, getline

i) open j) read k) seekg, seekp

14.2 a) 错误。函数 read 可以从 istream 派生的任何输入流对象读取。

b) 错误。这4个流自动生成。应在文件中包括 <iostream.h> 头文件,该文件首部包含了对这4个流对象的声明。

c) 错误。流对象离开范围或程序执行终止前执行 ifstream、ofstream 和 fstream 对象的析构函数时关闭文件,但作为一个编程习惯,应在文件不再需要时立即用 close 关闭。

d) 错误。成员函数 seekp 或 seekg 可以将 put 或 get 文件位置指针移到文件开头。

e) 正确。

f) 错误。大多数情况下,顺序文件的记录没有统一的长度。因此,更新一条记录可能会改写其他数据。

g) 正确。

h) 错误。随机访问文件中的记录通常具有统一的长度。

i) 错误。根据文件位置指针从文件起始点、结束点和当前点定位文件中的位置都是可能的。

14.3 a) ifstream inOldMaster("oldmast.dat", ios::in);

b) ifstream inTransaction("trans.dat", ios::in);

c) ofstream outNewMaster("newmast.dat", ios::out);

d) inOldMaster >> accountNum >> name >> currentBalance;

e) inTransaction >> accountNum >> dollarAmount;

o) outNewMaster << accountNum << name << currentBalance;

14.4 a) 错误:文件 payables.dat 尚未打开就试图向流输出数据。

改正:用 ostream 函数 open 打开 payables.dat 以便输出。

b) 错误:用错误的 istream 对象从文件“payables.dat”读取记录。

改正:用 istream 对象 inPayable 引用“payables.dat”。

c) 错误:删除了文件内容,因为文件打开便会输出(ios::out)。

改正:可以使用打开文件以便更新(ios::ate)或打开文件以便添加(os::app)的方法将数据添加到文件中。

### 练习题

14.5 填空题:

a) 计算机把大量的数据存储在二级存储设备(如\_\_\_\_\_)上。

- b) 一条\_\_\_\_\_由几个域组成。
- c) 可以包含数字、字母和空格的域称为\_\_\_\_\_域。
- d) 为了便于检索文件中的某条指定的记录,每条记录都有一个域被选作\_\_\_\_\_。
- e) 计算机系统的大多数信息存储在\_\_\_\_\_文件中。
- f) 表达了一定意义的一组相关的字符称为\_\_\_\_\_。
- g) 头文件 `<iostream.h>` 声明的标准流对象为\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- h) `ostream` 成员函数\_\_\_\_\_向流中写入一个字符。
- i) `ostream` 成员函数\_\_\_\_\_通常用于向随机访问文件中写入数据。
- j) `istream` 成员函数\_\_\_\_\_把文件位置指针重定位。

14.6 判断正误。如果有错,请说明原因。

- a) 从本质上说,计算机执行的是对 0 和 1 的操作。
- b) 人们更愿意对位进行操作,而不愿意操作字符或域,原因是位较为紧凑一些。
- c) 人们把程序和数据项表示成字符,然后计算机把这些字符作为 0 和 1 的组合进行操作和处理。
- d) 5 位邮政编码是一个数值域示例。
- e) 在计算机应用程序中,街道地址通常被视为字母组成的域。
- f) 计算机处理的数据项构成了数据的层次。在这个层次结构中,数据项按域、字符和位的顺序是越来越大,越来越复杂。
- g) 记录关键字属于某个特定的域,它能够识别一条记录。
- h) 为了便于计算机处理信息,多数机构都把他们的信息存储在一个文件中。C++ 程序总是通过名字来引用文件。
- j) 程序建立一个文件后,计算机会自动保存这个文件以便后期引用。

14.7 练习题 14.3 让读者编写了一组语句,这些语句实际上正好组成一类重要的文件处理程序(即文件匹配程序)的核心。在商业数据处理中,通常在每个系统中都有多个文件。例如,在应收账款系统中,一般有一个主文件,它包含了每个顾客的详细信息,如姓名、地址、电话号码、未付的欠款、信用额度、合同管理,还可能有近期购买和现金付款的简单记录。事务发生时(即货物卖出、汇款已邮到),这些信息被输入到一个文件中。每个商业周期(有的公司是一个月,有的是一个星期,还有的是一天)结束时,这个事务文件(在练习题 14.3 中称为“trans.dat”)用于主文件(在练习题 14.3 中称为 oldmast.dat)更新账户购买和付款记录。每更新一次后,主文件就被改写为一个新文件(newmast.dat),它用于在下一个商业周期快结束时执行更新过程。文件匹配程序必须处理一些单文件程序中不存在的问题。例如,并非总是发生匹配,主文件名某位顾客可能没有在目前的商业周期中购货或付款,这样在事务文件中就没有这个顾客的记录。类似地,某位购货或付款的客户可能刚搬到这个社区来,公司还来不及为这个顾客建立记录。以练习题 14.3 中的语句为基础,编写一个完整的文件匹配应收账款程序。为了匹配,可以用每个文件上的账号作为记录关键字。假设每个文件都是顺序文件,记录是按账号递增的顺序存储的。当发生匹配时(即具有相同账号的记录在主文

件和事务文件中同时出现),把事务文件上的美元数加到主文件的当前结算额上,并把记录写入 newmast.dat 中(假定购货在事务文件中用正数表示,付款在文件中用负数表示)。当某个特定的账户只有主记录但没有对应的事务记录时,只把主记录写入 newmast.dat 中。当只有事务记录而没有对应的主记录时,打印出消息“Unmatched transaction record for account number...”(在省略号处填入事务记录的账号)。

- 14.8 编写好练习题 4.7 中的程序后,编写一个简单程序建立一组检验数据,使用如下的示范数据测试练习题 14.7 中的程序:

| 主文件 |            |         |
|-----|------------|---------|
| 账号  | 姓名         | 结算额(美元) |
| 100 | Alan Jones | 348.17  |
| 300 | Mary Smith | 27.19   |
| 500 | Sam Sharp  | 0.00    |
| 700 | Suzy Green | -14.22  |

| 事务文件 |         |
|------|---------|
| 账号   | 交易额(美元) |
| 100  | 27.14   |
| 300  | 62.11   |
| 400  | 100.56  |
| 900  | 82.17   |

- 14.9 用练习题 14.8 中建立的检验数据文件运行练习题 14.7 中的程序,用 14.7 节的程序打印一个新的主文件,仔细检查结果。
- 14.10 有时几个事务记录的记录关键字相同,这非常普遍,因为一个顾客在同一个商业周期内可能多次购货或付款。重写练习题 14.7 中的应收账款文件匹配程序,使它能够处理具有相同关键字的多个事务记录。修改练习题 14.8 中的检验数据,使其包含以下的事务记录:

| 账号  | 数额(美元) |
|-----|--------|
| 300 | 83.89  |
| 700 | 80.78  |
| 700 | 1.53   |

- 14.11 写出符合下列要求的语句。假定已经定义了下面的结构并打开了用于写入数据的随机访问文件。

```
struct person{
    char lastName[15];
    char firstName[15];
    char age[2];
};
```

- a) 初始化文件 nameage.dat,使其拥有 100 个 lastName = "unsigned", firstName = "",

age = "0" 的记录。

b) 输入 10 个姓、名和年龄,并将它们写入文件。

c) 更新其中已有信息的文件,如果没有信息则告诉用户“没有信息”。

d) 删除一条已有信息的记录(可以重新初始化这条记录)。

- 14.12 你是一家五金商店的老板。为了查看工具种类、工具数量以及每件工具的价格,需要编制一份商品目录。编写一个程序,把文件 hardware.dat 初始化为 100 条空记录,输入每件工具的有关数据,能够列出所有工具的清单、删除某个工具不存在的记录以及更新文件中的任何信息。用工具标识号作为记录号,使用下列信息开始文件:

| 记录号# | 工具名称            | 数量  | 价值(美元) |
|------|-----------------|-----|--------|
| 3    | Electric sander | 7   | 57.98  |
| 17   | Hammer          | 76  | 11.99  |
| 24   | Jig saw         | 21  | 11.00  |
| 39   | Lawn mower      | 3   | 79.50  |
| 56   | Power saw       | 18  | 99.99  |
| 68   | Screwdriver     | 106 | 6.99   |
| 77   | Sledge hammer   | 11  | 21.50  |
| 83   | Wrench          | 34  | 7.50   |

- 14.13 修改第 4 章编写的电话号码字生成程序,使其将输出写入一个文件,以便阅读文件。如果有可用的计算机专业词典,可以将程序修改成查找词典中的 7 字母单词。这个程序产生一些有趣的 7 字母单词组合,该组合可能包括两个或 3 个单词。例如,电话号码 8432677 会产生“THEBOSS”。将程序修改成用计算机专业词典查找词典中的 7 字母单词,看看是否有一个字母的单词加 6 个字母的单词、两个字母的单词加 5 个字母的单词等等。

- 14.14 编写一个程序,用 sizeof 操作符确定计算机系统中各种数据类型占用的字节数。为便于以后的打印,把结果写入文件 datasize.dat。文件中的信息格式为:

| 数据类型               | 大小 |
|--------------------|----|
| char               | 1  |
| unsigned char      | 1  |
| short int          | 2  |
| unsigned short int | 2  |
| int                | 4  |
| unsigned long int  | 4  |
| float              | 4  |
| double             | 8  |
| long double        | 16 |

说明:读者计算机系统的数据类型大小可能与上面列出的不一样。

# 第 15 章 数据结构

## 学习目标

- 能够使用指针、自引用类和递归形成链接数据结构
- 能够创建和操纵动态数据结构,如链表、队列、堆栈及二叉树
- 了解链接数据结构的各种重要应用
- 了解如何使用类模板、继承和构造创建可重用的数据结构

## 15.1 简介

我们已经学习了固定大小的数据结构如 一维数组,二维数组和结构(struct)。本章将介绍在程序执行时可动态增长或缩小的“动态数据结构”。“链表”是“排列成一行”的数据项的集合,其插入与删除操作可在其任何一处进行。“堆栈”在编译器与操作系统中的地位非常重要,其插入与删除操作只能在其顶部进行。“队列”相当于排队,其插入操作在尾部进行,删除操作在头部进行。“二叉树”在数据快速查找、排序及重复数据项删除中尤为有效,其典型应用为文件系统目录管理及将表达式翻译为机器语言。这些数据结构还有很多有趣的应用。

接下来我们将讨论这几种主要的数据结构并用程序实现和操纵这些数据结构。为了提高可重用性与可维护性,我们用类、类模板、继承和构造来创建、包装这些数据结构。

本章的学习将为第 20 章“标准模板库(STL)”的学习作铺垫。标准模板库(STL)是C++标准库中的主要组成部分。它提供了容器,遍历容器的迭代器和处理容器中元素的算法。你可以看到标准模板库使用了本章中讨论的每一种数据结构并将它们打包成模板类。标准模板库中的代码经过精心设计而具有较高效率,良好的可移植性及可扩充性。一旦理解了本章所述数据结构的主要规则及如何创建数据结构之后,你将能充分地利用标准模板库中经过包装了的数据结构、迭代器和算法。标准模板库是迄今为止人们对C++标准最大的增强。它是帮助实现重用、再重用、再重用思想的全世界通用的组件集合。

你可将本章中的例程用于更高级的课程和工业实际应用中。这些例程大量使用了指针。同时练习题中也包含了丰富的例子。

我们鼓励你大胆尝试标题为“构建自己的编译器”特色部分中所描述的内容。你可能正在用编译器将自己的C++程序翻译为机器代码以便能在自己的机器上执行它们。在这个工程中,需要自行构建编译器。它会从文件中读取用与流行的 BASIC 语言的早期版本类似虽简单但功能强大的高级语言所编写的程序。你的编译器会将这些源程序翻译成由简单机器语言(简称 SML,有关该语言的详情,参见第 5 章特色部分“建立自己的计算机”)指令所组成的文件。你的 Simpletron 模拟器将会执行这些由编译器产生的 SML 程序。大量使用面向

对象的方法来实现这个工程将给你一个宝贵的机会,以实践你在本课程中所学的大部分内容。特色部分在引导你逐步了解高级语言的规范的同时,还描述了将高级语言源程序转换为机器指令的算法。如果你乐于接受挑战,还可以尝试练习中对 Simpletron 模拟器和编译器进行扩展了的内容。

## 15.2 自引用类

自引用类包含一个指针成员,它指向与其具有相同类型的对象。如类定义

```
class Node {
public:
    Node(int);
    void setData(int);
    int getData() const;
    void setNextPtr(Node *);
    const Node *getNextPtr() const;
private:
    int data;
    Node *nextPtr;
};
```

定义了一个 Node 类。Node 类带有两个 private 数据成员:整型成员 data 和指针成员 nextPtr。成员 nextPtr 指向同样是 Node 类的一个对象,因此 Node 类就被称之为“自引用类”。成员 nextPtr 表示一个链接,也就说它可被用来将一个 Node 类对象与另一个 Node 类对象链接在一起。Node 类同时也有 5 个成员函数:构造函数用来接收一个整型值并且初始化数据成员 data;函数 setData 用来设定数据成员 data 的值;函数 getData 返回成员 data 的值;函数 setNextPtr 用来设置成员 nextPtr 的值;函数 getNextPtr 返回数据成员 nextPtr 的值。

自引用类对象能够被链接起来而形成有用的数据结构如链表、队列、堆栈和树。图 15.1 为两个自引用类的对象被链接在一起而形成链表。注意到第二个对象的指针成员中的短斜线,它是一个空指针,表示第二个对象没指向任何对象。短斜线只是作例示用的,并不代表 C++ 中的字符“\”。空指针通常表示数据结构的尾端,如同空字符('\0')表示字符串结束一样。

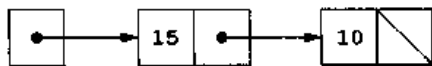


图 15.1 链接在一起的两个自引用类对象

常见编程错误 15.1 未将链表最后一个节点的指针置为空。

## 15.3 动态内存分配

创建与维持动态数据结构需要动态内存分配,即程序可以获得更多的内存以容纳新的节点和释放数据结构不再需要的内存。动态内存分配大小的限制与机器可使用的物理内存相当或与虚拟内存系统中的可获得的虚拟内存相当。但是,这种大小的限制较小,因为内存

供多个用户共享。

在动态内存分配中要用到操作符 `new` 和 `delete`, 操作符 `new` 接收被分配的类对象作为变量, 同时一个返回指向此类对象的指针。例如语句

```
Node *newPtr = new Node(10);
```

会分配 `sizeof(Node)` 字节的内存, 同时调用构造函数并将指针存储到变量 `newPtr` 中。如果没有内存可供分配, `new` 会抛出异常 `bad_alloc`。语句中的数字 10 代表传给节点对象的数据。

`delete` 操作符调用析构函数并且回收操作符 `new` 所分配的内存, 以便系统能够再进行内存分配。要释放操作符 `new` 分配的内存, 可用语句

```
delete newPtr;
```

注意 `newPtr` 本身并没有被删除, 仅仅是它所指向的空间被释放了。如果 `newPtr` 的值为 0 (表示不指向任何对象), 以上语句就没起到任何作用。

随后将讨论链表、堆栈、队列和树。这些数据结构是用动态内存分配和自引用类来创建和维持的。

**可移植性提示 15.1** 类对象的大小不一定是其数据成员大小的总和, 这是因为有的机器要求存储内容必须沿存储单元的边界开始存储, 当然还有别的原因 (参见第 16 章)。因此尽量用操作符 `sizeof` 来确定对象的大小。

**常见编程错误 15.2** 认为类对象的大小等于其数据成员大小的总和。

**常见编程错误 15.3** 没有释放不再需要的动态分配内存会导致系统过早耗尽内存, 这有时被称为“内存泄漏”。

**良好编程习惯 15.1** 不再需要操作符 `new` 分配的内存时, 立刻用 `delete` 释放内存。

**常见编程错误 15.4** 用操作符 `delete` 释放不是用操作符 `new` 动态分配的内存。

**常见编程错误 15.5** 引用被删除的内存。

**常见编程错误 15.6** 试图删除已被删除的内存可能会在运行时造成不可预料的结果。

## 15.4 链表

“链表”是用指针进行链接的自引用类对象 (也就是“节点”) 的线性集合, 其名称也由此而来。链表的访问是通过指向其第一个节点的指针来实现的, 随后的节点可通过存储在每个节点中的指针成员来访问。按照规定, 链表中的最后一个节点的指针成员应置为空 (即置为 0) 以表示链表结束。一个节点可以包含任意类型的数据, 包括其他类的对象。如果节点包括基类指针或基类引用以及派生类对象, 我们就可以调用虚拟函数来对这些对象进行多态处理。堆栈与队列也是线性的数据结构, 我们也可以看到它们实际上是链表增加了一些限制的变体。树属于非线性的数据结构。

列表数据可以存储在数组中, 但使用链表提供了几种好处。当数据结构中的数据成员的数目无法预知的时候, 链表便可发挥其作用。链表是动态的, 因此其长度可以随需要而增加或减小。而常规 C++ 数组的大小是无法改变的, 因为其大小在编译时已经确定。常规数



组也可能为满,而链表只有在系统的内存不足时才会变满。

**性能提示 15.1** 在声明时,可以把数组元素的个数定义为多于数据项数,但这样会浪费内存。此时,链表能够更有效地利用内存,它允许程序在运行时进行调整。

可以在链表的适当位置插入新的节点而不破坏有序链表的有序状态,已有的节点元素无需任何移动。

**性能提示 15.2** 数组的插入与删除操作可能很耗时间,因为插入点和删除点之后的所有元素都要作相应移动。

**性能提示 15.3** 数组元素在内存中是连续存放的,这样一来,就可直接存取数组元素,因为任何数组元素的地址都可以直接通过它与数组中第一个元素的相对位置来确定。而链表不能直接存取其中的元素。

链表节点在内存中一般不是连续存放的,但从逻辑上讲,链表的节点看起来却是连续的。图 15.2 为拥有几个节点的链表。

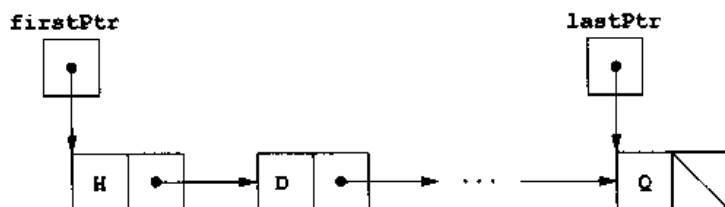


图 15.2 链表图示

**性能提示 15.4** 对于那些运行时会动态增长或缩小的数据结构,如用动态内存分配(而不是数组)来处理可节约存储空间。但要记住,指针也会占用空间,而且动态内存分配会引起调用函数的开销。

图 15.3 中的程序(其输出结果如图 15.4 所示)使用了类模板 List(参见第 12 章)来操作—列整型数据和—列浮点数据。主程序(fig15\_03.cpp)提供了 5 个选项:

- (1) 在表头插入一个数据;
- (2) 在表尾插入一个数据;
- (3) 从表头删除一个数据;
- (4) 从表尾删除一个数据;
- (5) 终止程序。

```

1 //Fig.15.3: listnd.h
2 //ListNode template definition
3 #ifndef LISTND_H
4 #define LISTND_H
5
6 template< class NODETYPE > class List; //forward declaration
7
8 template<class NODETYPE>
9 class ListNode {
10     friend class List< NODETYPE >; //make List a friend
  
```

```

11 public:
12     ListNode( const NODETYPE & ); //constructor
13     NODETYPE getData() const; //return data in the node
14 private:
15     NODETYPE data;                //data
16     ListNode< NODETYPE > *nextPtr; //next node in the list
17 };
18
19 //Constructor
20 template<class NODETYPE>
21 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
22     : data( info ), nextPtr( 0 ) {}
23
24 //Return a copy of the data in the node
25 template< class NODETYPE >
26 NODETYPE ListNode< NODETYPE >::getData() const { return data; }
27
28 #endif

```

图 15.3 操作链表——listnd.h

```

29 //Fig.15.3: list.h
30 //Template List class definition
31 #ifndef LIST_H
32 #define LIST_H
33
34 #include <iostream>
35 #include <cassert>
36 #include "listnd.h"
37
38 using std::cout;
39
40 template< class NODETYPE >
41 class List {
42 public:
43     List();           //constructor
44     ~List();          //destructor
45     void insertAtFront( const NODETYPE & );
46     void insertAtBack( const NODETYPE & );
47     bool removeFromFront( NODETYPE & );
48     bool removeFromBack( NODETYPE & );
49     bool isEmpty() const;
50     void print() const;
51 private:
52     ListNode< NODETYPE > *firstPtr; //pointer to first node
53     ListNode< NODETYPE > *lastPtr;  //pointer to last node
54
55     //Utility function to allocate a new node
56     ListNode< NODETYPE > *getNode( const NODETYPE & );
57 };
58

```

```
59 //Default constructor
60 template< class NODETYPE >
61 List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) {}
62
63 //Destructor
64 template< class NODETYPE >
65 List< NODETYPE >::~~List()
66 {
67     if ( ! isEmpty() ) { //List is not empty
68         cout << "Destroying nodes ... \n";
69
70         ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;
71
72         while ( currentPtr != 0 ) { //delete remaining nodes
73             tempPtr = currentPtr;
74             cout << tempPtr->data << ' \n';
75             currentPtr = currentPtr->nextPtr;
76             delete tempPtr;
77         }
78     }
79
80     cout << "All nodes destroyed \n \n";
81 }
82
83 //Insert a node at the front of the list
84 template< class NODETYPE >
85 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
86 {
87     ListNode< NODETYPE > *newPtr = getNewNode( value );
88
89     if ( isEmpty() ) //List is empty
90         firstPtr = lastPtr = newPtr;
91     else {           //List is not empty
92         newPtr->nextPtr = firstPtr;
93         firstPtr = newPtr;
94     }
95 }
96
97 //Insert a node at the back of the list
98 template< class NODETYPE >
99 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
100 {
101     ListNode< NODETYPE > *newPtr = getNewNode( value );
102
103     if ( isEmpty() ) //List is empty
104         firstPtr = lastPtr = newPtr;
105     else {           //List is not empty
106         lastPtr->nextPtr = newPtr;
107         lastPtr = newPtr;
108     }
109 }
```

```

110
111 //Delete a node from the front of the list
112 template< class NODETYPE >
113 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
114 {
115     if ( isEmpty() )           //List is empty
116         return false;          //delete unsuccessful
117     else {
118         ListNode< NODETYPE > *tempPtr = firstPtr;
119
120         if ( firstPtr == lastPtr )
121             firstPtr = lastPtr = 0;
122         else
123             firstPtr = firstPtr->nextPtr;
124
125         value = tempPtr->data; //data being removed
126         delete tempPtr;
127         return true;          //delete successful
128     }
129 }
130
131 //Delete a node from the back of the list
132 template< class NODETYPE >
133 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
134 {
135     if ( isEmpty() )
136         return false; //delete unsuccessful
137     else {
138         ListNode< NODETYPE > *tempPtr = lastPtr;
139
140         if ( firstPtr == lastPtr )
141             firstPtr = lastPtr = 0;
142         else {
143             ListNode< NODETYPE > *currentPtr = firstPtr;
144
145             while ( currentPtr->nextPtr != lastPtr )
146                 currentPtr = currentPtr->nextPtr;
147
148             lastPtr = currentPtr;
149             currentPtr->nextPtr = 0;
150         }
151
152         value = tempPtr->data;
153         delete tempPtr;
154         return true; //delete successful
155     }
156 }
157
158 //Is the List empty?
159 template< class NODETYPE >
160 bool List< NODETYPE >::isEmpty() const

```

```

161     | return firstPtr == 0; }
162
163 //Return a pointer to a newly allocated node
164 template< class NODETYPE >
165 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
166     const NODETYPE &value )
167 {
168     ListNode< NODETYPE > *ptr =
169     new ListNode< NODETYPE >( value );
170     assert( ptr != 0 );
171     return ptr;
172 }
173
174 //Display the contents of the List
175 template< class NODETYPE >
176 void List< NODETYPE >::print() const
177 {
178     if ( isEmpty() ) {
179         cout << "The list is empty\n\n";
180         return;
181     }
182
183     ListNode< NODETYPE > *currentPtr = firstPtr;
184
185     cout << "The list is: ";
186
187     while ( currentPtr != 0 ) {
188         cout << currentPtr->data << " ";
189         currentPtr = currentPtr->nextPtr;
190     }
191
192     cout << "\n\n";
193 }
194
195 #endif

```

图 15.3 操作链表——list.h

```

196 //Fig. 15.3; fig15_03.cpp
197 //List class test
198 #include <iostream>
199 #include "list.h"
200
201 using std::cin;
202 using std::endl;
203
204 //Function to test an integer List
205 template< class T >
206 void testList( List< T > &listObject, const char *type )
207 {
208     cout << "Testing a List of " << type << " values\n";

```

```
209
210     instructions();
211     int choice;
212     T value;
213
214     do {
215         cout << "? ";
216         cin >> choice;
217
218         switch ( choice ) {
219             case 1:
220                 cout << "Enter " << type << ": ";
221                 cin >> value;
222                 listObject.insertAtFront( value );
223                 listObject.print();
224                 break;
225             case 2:
226                 cout << "Enter " << type << ": ";
227                 cin >> value;
228                 listObject.insertAtBack( value );
229                 listObject.print();
230                 break;
231             case 3:
232                 if ( listObject.removeFromFront( value ) )
233                     cout << value << " removed from list\n";
234
235                 listObject.print();
236                 break;
237             case 4:
238                 if ( listObject.removeFromBack( value ) )
239                     cout << value << " removed from list\n";
240
241                 listObject.print();
242                 break;
243         }
244         while ( choice != 5 );
245
246         cout << "End list test\n\n";
247     }
248
249     void instructions()
250     {
251         cout << "Enter one of the following:\n"
252              << " 1 to insert at beginning of list\n"
253              << " 2 to insert at end of list\n"
254              << " 3 to delete from beginning of list\n"
255              << " 4 to delete from end of list\n"
256              << " 5 to end list processing\n";
257     }
258
259     int main()
```

```
260 {  
261     List< int > integerList;  
262     testList( integerList, "integer" ); //test integerList  
263  
264     List< double > doubleList;  
265     testList( doubleList, "double" ); //test doubleList  
266  
267     return 0;  
268 }
```

图 15.3 操作链表

输出结果:

```
Testing a List of integer values  
Enter one of the following:  
  1 to insert at beginning of list  
  2 to insert at end of list  
  3 to delete from beginning of list  
  4 to delete from end of list  
  5 to end list processing  
? 1  
Enter integer:1  
The list is:1  
  
? 1  
Enter integer:2  
The list is:2 1  
  
? 2  
Enter integer:3  
The list is ;2 1 3  
  
? 2  
Enter integer:4  
The list is ;2 1 3 4  
  
? 3  
2 removed from list  
The list is:1 3 4  
  
? 3  
1 removed from list  
The list is:3 4  
  
? 4  
4 removed from list  
The list is:3  
  
? 4  
3 removed from list  
The list is empty
```

```
? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 6
Enter double:1.1
The list is:1.1

? 1
Enter double:2.2
The list is:2.2 1.1

? 2
Enter double:3.3
The list is:2.2 1.1 3.3

? 2
Enter double:4.4
The list is:2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is:1.1 3.3 4.4

? 3
1.1 removed from list
The list is:3.3 4.4

? 4
4.4 removed from list
The list is:3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

图 15.4 图 15.3 所示程序的输出结果



稍后将详细讨论这些程序。练习题 15.20 要求你实现一个递归函数逆向打印, 练习题 15.21 要求你实现一个循环函数查找指定的数据项。

图 15.3 包含了两个类模板: `ListNode` 与 `List`。每个 `List` 对象中封装了一些互相链接的 `ListNode` 对象, `ListNode` 类模板包括两个 `private` 数据成员 `data` 和 `nextPtr`。 `ListNode` 数据成员 `data` 存储一个 `NODETYPE` 类型的数据, 类型参数被传递给类模板; 数据成员 `nextPtr` 存储了指向链表中下一个 `ListNode` 对象的指针。

`List` 类模板包含 `private` 成员 `firstPtr` (指向第一个 `ListNode` 对象) 与 `lastPtr` (指向最后一个 `ListNode` 对象), 默认的构造函数将这两个成员初始化为空 (即 0)。析构函数用于确保所有的 `ListNode` 对象能在 `List` 对象被删除时删除。 `List` 类模板主要的成员函数包括 `insertAtFront`, `insertAtBack`, `removeFromFront` 和 `removeFromBack`。

函数 `isEmpty` 被称为判断函数, 因为它并不改变 `List` 对象, 而只是确定 `List` 对象是否为空 (即确定 `List` 对象指向第一个节点的指针是否为空)。空则返回 `true`, 反之则返回 `false`。函数 `print` 显示 `List` 对象所包含的内容。

**良好编程习惯 15.2** 把空值 (零) 赋值给链表新节点的成员, 使用指针前应进行初始化。

在随后的小节中, 将详细讨论类 `List` 的每个成员函数。函数 `insertAtFront` (参见图 15.5) 在链表头放置一个新的节点, 具体步骤如下:

(1) 调用函数 `getNode`, 并将变量 `value` 的常量引用值传递给它。

(2) 函数 `getNode` 使用操作符 `new` 创建一个新的节点, 同时返回指向这个节点的指针。如果此指针非空, `getNode` 则将此指针返回给函数 `insertAtFront` 中的指针变量 `newPtr`。

(3) 如果链表为空, 指针变量 `firstPtr` 与 `lastPtr` 就等于 `newPtr`。

(4) 如果链表非空, 则先将 `newPtr -> nextPtr` 指向 `firstPtr` 所指向的节点即将新节点指向原链表中的第一个节点, 然后令 `firstPtr` 指向新增加的节点即让 `firstPtr` 指向链表新的首节点。通过以上操作, `newPtr` 所指向的节点就被插入到了链表头部。

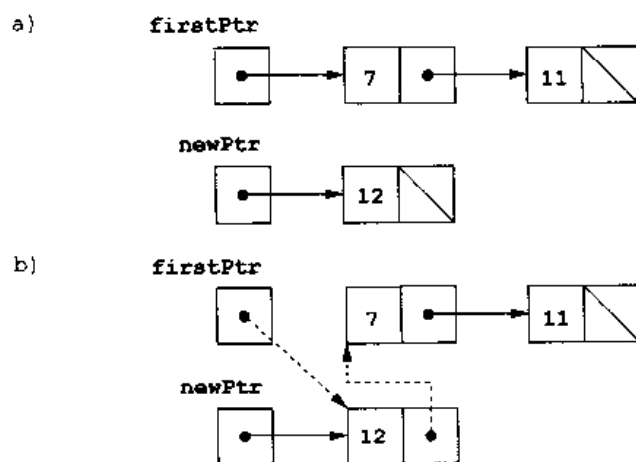


图 15.5 `insertAtFront` 的操作过程

图 15.5 演示了函数 `insertAtFront` 的操作过程, 其中的 a) 部分为 `insertAtFront` 操作前链

表与新节点的状态。而 b) 部分中的虚线箭头则演示了 insertAtFront 操作过程中的第 2 步和第 3 步, 这两个步骤将其中包含新节点 12 的节点插到新表头部。

函数 insertAtBack(如图 15.6 所示)用于在表尾增加新节点。其操作步骤如下:

- (1) 调用函数 getNode, 并将变量 value 的常量引用值传递给它。
- (2) 函数 getNode 使用操作符 new 创建一个新的节点, 同时返回指向这个节点的指针。如果此指针非空, getNode 则将此指针返回给函数 insertAtBack 中的指针变量 newPtr。
- (3) 如果链表为空, 则指针变量 firstPtr 与 lastPtr 等于 newPtr。
- (4) 如果链表非空, 则先将 lastPtr -> nextPtr 指向 newPtr 所指向的节点即令原链表中的最后一个节点指向新节点, 然后令 lastPtr 指向新增加的节点即让 lastPtr 指向链表新的尾节点。通过以上操作, newPtr 所指向的节点就被插入到了链表尾部。

图 15.6 演示了函数 insertAtBack 的操作过程, 其中 a) 为 insertAtBack 操作前链表与新节点的状态, 而 b) 中的虚线箭头则演示了将新节点插入到表尾的过程。

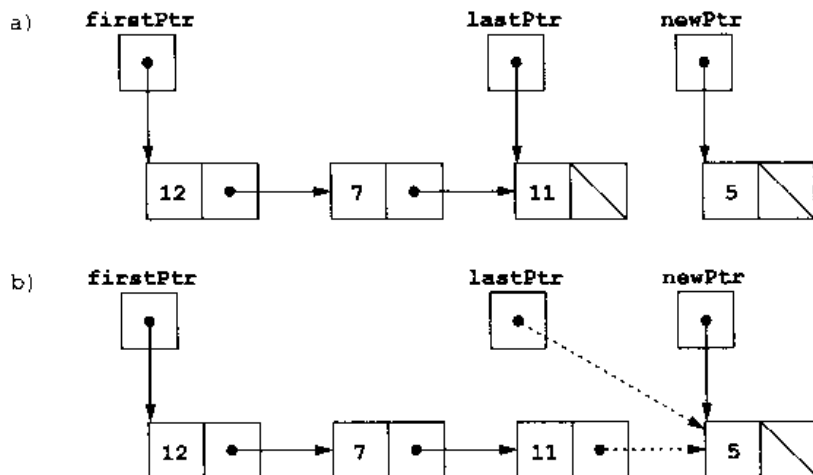
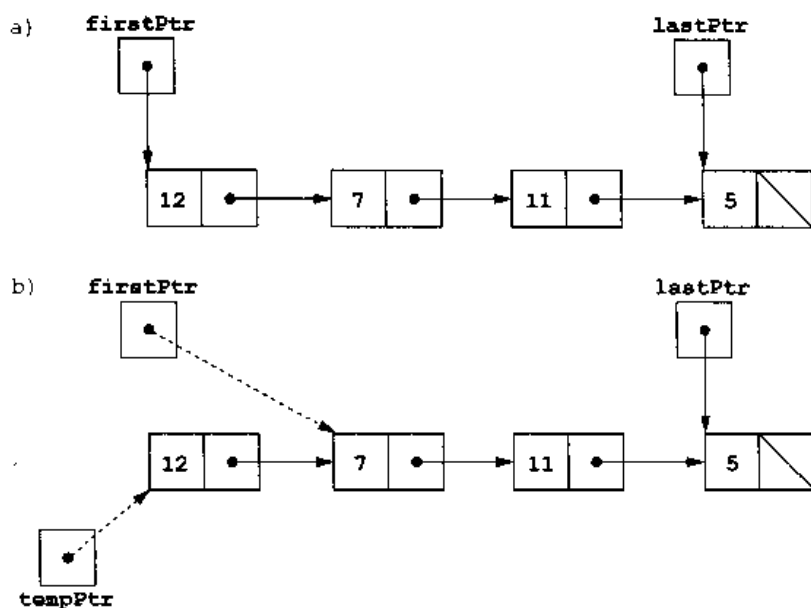


图 15.6 insertAtBack 的操作过程

函数 removeFromFront(如图 15.7 所示)从表头删除一个节点并将所删节点值传递给引用参数 value。操作成功, 函数将返回 true, 反之返回 false。其操作步骤如下:

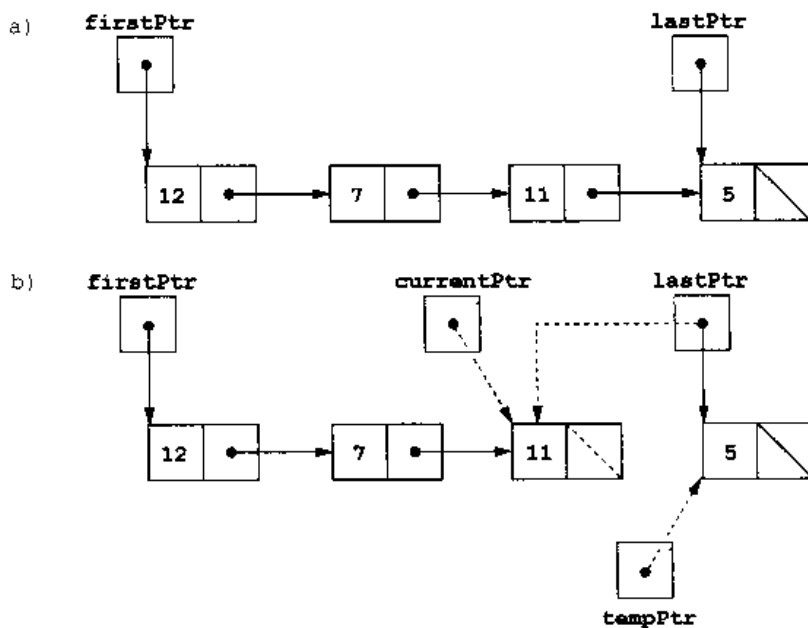
- (1) 将 firstPtr 的值赋给 tempPtr;
- (2) 如果 firstPtr 等于 lastPtr, 如在链表只有一个节点的情况下, 则将 firstPtr 与 lastPtr 置空并删除节点;
- (3) 如果链表节点不止一个, 则保持 lastPtr 不变, 而将 firstPtr 指向 firstPtr -> nextPtr 所指向的对象。即修改 firstPtr 使其指向节点删除前的链表中的第二个节点;
- (4) 当所有的指针操作完成后, 将待删除节点的数据成员 data 的值传递给引用参数 value;
- (5) 删除 tempPtr 所指向的节点;
- (6) 返回 true, 表明操作成功。

图 15.7 演示了函数 removeFromFront 的操作过程, 其中 a) 部分为操作前的状态, 而 b) 为实际的指针操作过程。

图 15.7 `removeFromFront` 的操作过程

函数 `removeFromBack` (如图 15.8 所示) 从表尾删除一个节点并将所删节点值传递给引用参数 `value`。操作成功, 函数将返回 `true`, 返回 `false`。其操作步骤如下:

- (1) 将 `lastPtr` 指向的地址值赋给 `tempPtr`, 最后用 `tempPtr` 删除打算删除的节点;
- (2) 如果 `firstPtr` 等于 `lastPtr`, 也就是说, 如执行删除操作前链表只有一个节点, 则将 `firstPtr` 与 `lastPtr` 设置为 0, 并将节点与链表断开 (令链表为空表);
- (3) 如果链表有多于一个的节点, 则先将 `firstPtr` 的地址赋给 `currentPtr`;
- (4) 现在随 `currentPtr` 遍历链表, 直到它指向倒数第二节点为止。这是通过 `while` 循环来实现的。While 循环会在 `currentPtr -> nextPtr` 不是 `lastPtr` 时, 不断地由 `currentPtr -> nextPtr`

图 15.8 `removeFromBack` 的操作过程

来代替 currentPtr;

- (5) 将 currentPtr 的值赋给 lastPtr;
- (6) 使 currentPtr -> nextPtr 为空;
- (7) 完成所有指针操作后,将待删除节点的数据成员 data 的值传递给引用参数 value;
- (8) 删除 tempPtr 所指向的节点;
- (9) 返回 true 表明操作成功。

图 15.8 演示了函数 removeFromBack 的操作过程,其中 a) 部分为操作前的状态,而 b) 部分为实际的指针操作过程。

函数 print 首先确定链表是否为空,如为空,则显示“The list is empty”并返回。如非空,则将依次显示链表中的节点数据。函数首先将 firstPtr 的值赋给 currentPtr,然后显示“The list is: ”。当 currentPtr 非空时,显示数据 currentPtr -> data,然后将 currentPtr -> nextPtr 的值赋给 currentPtr。值得注意的是,如果链表最后一个节点的指针非空时,此算法会错误显示链表之后的其他数据。此数据算法也适用于堆栈和队列。

以上所讨论的链表称为“单向链表”。这类链表以指向第一个节点的指针开始,然后每一个节点都包含一个指向链表中下一节点的指针,最后以一个指针成员为空的节点结束链表。单向链表只能按一个方向来遍历。

下面将简单讨论一下其他几种链表:循环单向链表、双向链表和循环双向链表。

同单向链表一样,循环单向链表也以指向第一个节点的指针开始,然后每一个节点都包含一个指向链表中下一节点的指针,但不同的是它的最后一个节点的指针成员不为空,而是指向第一个节点,因此被称为循环单向链表。

双向链表允许前后两个方向的链表遍历。这种链表通常有两个起始指针,一个指向第一个节点以用于由前向后的链表遍历,另一个指向最后一个节点以用于由后向前的链表遍历。而且此种链表中每个节点都有两个指针,一个指向之前的节点,另一个指向之后的节点。例如,你的链表包含按字母排序的电话号码簿,此时如果你想查找一个姓名比较靠前的记录,你可采取由前到后的遍历次序,但要是你要查找的是一个姓名比较靠后的记录,你便可采取由后到前的遍历次序。

在循环双向链表中,最后一个节点向后的指针指向第一个节点,而第一个节点向前的指针则指向最后一个节点。

## 15.5 堆栈

第 12 章介绍了用基础数组实现方法的堆栈类模板。这里将介绍利用基础指针链表来实现方法的堆栈类模板。第 20 章将继续讨论堆栈。

“堆栈”是一种受限制的链表——节点的增加、删除只能从栈顶进行。正由于此,堆栈也被称为“后进先出”(LIFO)的数据结构。堆栈最后一个节点的链接成员被设为表示栈底的空(零)。

**常见编程错误 15.7** 未将栈底节点的链接成员设为空(零)。

用于操作堆栈的主要成员函数有 push 和 pop。函数 push 在栈顶压入一个新节点,函数

pop 从栈顶弹出一个节点,并将节点值传递给引用参数从而可以返回给调用函数,如果 pop 操作成功则返回 true,否则返回 false。

堆栈有很多有趣的应用。当调用函数时,被调用的函数要知道如何返回调用它的函数,此时返回地址便可压入堆栈中。在发生一系列函数调用的情况下,它们的返回地址依次被压入堆栈中,以便每一个函数能够正确返回其调用者。堆栈支持常规的函数调用,同时也支持递归函数调用,调用方式和常规的非递归函数调用一样。

堆栈也包含了每次调用函数时为其自动变量创建的空间。当函数返回其调用者或抛出异常时,就会调用每个局部对象的析构函数(如果有的话),为这些自动变量创建的空间就会从堆栈中弹出,这些自动变量也不再用于程序。

堆栈同样可用于编译器,供其计算表达式和产生机器语言代码。练习题中包含堆栈的几种应用,其中之一就是创建一个完整的、可工作的编译器。

我们将利用链表与堆栈的密切关系,通过重用链表类来创建一个堆栈类。这里我们使用了两种不同的重用方式。其中之一是通过 private 方式继承链表类来实现,另外一种是通过对链表类作为堆栈类的 private 成员来达到同一目的。你可以注意到本章中所有的数据结构包括下而将要谈到的两个堆栈类,都是通过模板来实现的,目的是进一步提高可重用性。

图 15.9 的程序(演示输出如图 15.10)通过 private 方式继承图 15.3 中的类模板 List 来实现一个堆栈类。我们让这个堆栈类拥有 4 个成员函数:push, pop, isEmpty 和 printStack。它们要用到类模板 List 的几个成员函数:insertAtFront, removeFromFront, isEmpty 和 print。当然类模板还包括其他的成员函数(如 insertAtBack 和 removeFromBack),但在这里我们不希望它们能够通过堆栈类 Stack 的 public 接口进行访问。因此我们指定堆栈类 Stack 以 private 方式继承链表类 List,这就使得类模板 List 所有的成员函数只能在类模板 Stack 内访问。然后通过调用 List 类相应的函数来实现类模板 Stack 的成员函数:push 调用 insertFromFront;pop 调用 removeFromFront;isEmpty 调用 isEmpty;printStack 调用 print。

```

1 //Fig.15.9: stack.h
2 //Stack class template definition
3 //Derived from class List
4 #ifndef STACK_H
5 #define STACK_H
6
7 #include "list.h"
8
9 template< class STACKTYPE >
10 class Stack : private List< STACKTYPE > {
11 public:
12     void push( const STACKTYPE &d ) { insertAtFront( d ); }
13     bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
14     bool isEmpty() const { return isEmpty(); }
15     void printStack() const { print(); }
16 };
17
18 #endif

```

图 15.9 一个简单的堆栈程序——stack.h

```

19 //Fig. 15.9: fig15_09.cpp
20 //Driver to test the template Stack class
21 #include <iostream>
22 #include "stack.h"
23
24 using std::endl;
25
26 int main()
27 {
28     Stack< int > intStack;
29     int popInteger, i;
30     cout << "processing an integer Stack" << endl;
31
32     for ( i = 0; i < 4; i++ ) {
33         intStack.push( i );
34         intStack.printStack();
35     }
36
37     while ( ! intStack.isStackEmpty() ) {
38         intStack.pop( popInteger );
39         cout << popInteger << " popped from stack" << endl;
40         intStack.printStack();
41     }
42
43     Stack< double > doubleStack;
44     double val = 1.1, popdouble;
45     cout << "processing a double Stack" << endl;
46
47     for ( i = 0; i < 4; i++ ) {
48         doubleStack.push( val );
49         doubleStack.printStack();
50         val += 1.1;
51     }
52
53     while ( ! doubleStack.isStackEmpty() ) {
54         doubleStack.pop( popdouble );
55         cout << popdouble << " popped from stack" << endl;
56         doubleStack.printStack();
57     }
58     return 0;
59 }

```

图 15.10 一个简单的堆栈程序——fig15\_09.cpp

输出结果:

```

processing an integer Stack
The list is:0

```

```

The list is:1 0

```

```

The list is:2 1 0

```

```
The list is:3 2 1 0

3 popped from stack
The list is:2 1 0

2 popped from stack
The list is:1 0

1 popped from stack
The list is:0

0 popped from stack
The list is empty

processing a double Stack
The list is:1.1

The list is:2.2 1.1

The list is:3.3 2.2 1.1

The list is:4.4 3.3 2.2 1.1

4. popped from stack
The list is:3.3 2.2 1.1

3.3 popped from stack
The list is:2.2 1.1

2.2 popped from stack
The list is:1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

图 15.10 图 15.9 所示程序的输出结果

在主函数 main 中,堆栈类模板用于生成 Stack <int> 的一个实例 intStack。整型值从 0 到 3 以先进后出的次序依次压入堆栈 intStack 中后又依次弹出。堆栈类模板也用于生成 Stack <double> 的一个实例 doubleStack。双精度值 1.1,2.2,3.3,4.4 同样以先进后出的次序依次压入堆栈 doubleStack 中后又依次弹出。

另一种实现堆栈类模板的方法是重用 List 类模板。图 15.11 中的程序使用了程序 List 的两个头文件 list.h 和 listnd.h。它使用与前一种方法相同的主程序,不同的是用新的头文件 stack\_c.h 代替了 stack.h,程序的输出也相同。在此种方法中,堆栈类模板 Stack 在定义时将 List <STACKTYPE> 类型的对象 s 作为自己的数据成员。

```

1 //Fig. 15.11; stack_c.h
2 //Definition of Stack class composed of List object
3 #ifndef STACK_C
4 #define STACK_C
5 #include "list.h"
6
7 template< class STACKTYPE >
8 class Stack {
9 public:
10     //no constructor; List constructor does initialization
11     void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
12     bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
13     bool isEmpty() const { return s.isEmpty(); }
14     void printStack() const { s.print(); }
15 private:
16     List< STACKTYPE > s;
17 };
18
19 #endif

```

图 15.11 使用合成的简单堆栈程序——stack\_c.h

## 15.6 队列

“队列”类似于超市里等候付款的队伍：排在第一的人首先获得服务，后来的顾客只能排在队尾等候服务。队列节点只能从队列头部删除，新节点则只能从队列尾部增加。因此，队列被称为“先进先出”(FIFO)的数据结构。增添与删除节点的函数分别为 enqueue 和 dequeue。

队列在计算机系统中的应用非常广泛。大部分计算机只有一个处理器，因此一次只能有一个用户可以得到服务，其他用户则要在队列中登记。随着用户依次获得服务，用户的登记项也逐步向前移。排在队列最前面的登记项是下一个获得服务的对象。

打印缓冲也要用到队列。在一个多用户的环境中可能只有一台打印机，多个用户可能都在输出需要打印的数据。如果打印机忙，它们就以队列的形式缓冲到磁盘上（如同线缠绕到线轴上）直到打印机可用为止。

计算机网络中的信息包也是放于队列中的。每次一个包到达一个节点时，它应该沿着到达包目的地址的路径被路由到下一个节点。但路由节点一次只能处理一个包，其他的包置于队列中等待路由。

计算机网络中的文件服务器要处理多个网络用户的文件访问，但它的处理能力是有限的，所以如果用户所请求的文件访问超出其处理能力，这些请求就会置于队列中。

**常见编程错误 15.8** 未把队列中最后一个节点的指针设置为空(零)。

图 15.12(输出结果如图 15.13)通过 private 方式继承图 15.3 中的 List 类模板，创建了一个队列类模板 Queue。我们同时让它拥有 4 个成员函数：enqueue, dequeue, isEmpty, printQueue。我们注意到它们要用到类模板 List 的成员函数 insertAtBack, removeFromFront,



isEmpty 和 print。当然,类模板还包括其他成员函数(如 insertAtFront 和 removeFromFront),但在这里我们不希望它们能够通过堆栈类 Stack 的 public 接口访问。因此我们指定队列类 Queue 以 private 方式继承链表类 List,这就使得类模板 List 所有的成员函数只能在队列模板 Queue 内访问。然后通过调用类 List 相应的函数来实现类模板 Queue 的成员函数:enqueue 调用 insertAtBack; dequeue 调用 removeFromFront; isEmpty 调用 isEmpty; printStack 调用 print。

```

1 //Fig.15.12: queue.h
2 //Queue class template definition
3 //Derived from class List
4 #ifndef QUEUE_H
5 #define QUEUE_H
6
7 #include "list.h"
8
9 template < class QUEUETYPE >
10 class Queue: private List < QUEUETYPE > {
11 public:
12     void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
13     bool dequeue( QUEUETYPE &d )
14         { return removeFromFront( d ); }
15     bool isEmpty() const { return isEmpty(); }
16     void printQueue() const { print(); }
17 };
18
19 #endif

```

图 15.12 处理队列——queue.h

```

20 //Fig.15.12: fig15_12.cpp
21 //Driver to test the template Queue class
22 #include <iostream>
23 #include "queue.h"
24
25 using std::endl;
26
27 int main()
28 {
29     Queue< int > intQueue;
30     int dequeueInteger, i;
31     cout << "processing an integer Queue" << endl;
32
33     for ( i = 0; i < 4; i++ ) {
34         intQueue.enqueue( i );
35         intQueue.printQueue();
36     }
37
38     while ( ! intQueue.isEmpty() ) {
39         intQueue.dequeue( dequeueInteger );
40         cout << dequeueInteger << " dequeued" << endl;

```

```

41     intQueue.printQueue();
42     |
43
44     Queue< double> doubleQueue;
45     double val = 1.1, dequeuedouble;
46
47     cout << "processing a double Queue" << endl;
48
49     for ( i = 0; i < 4; i++ ) {
50         doubleQueue.enqueue( val );
51         doubleQueue.printQueue();
52         val += 1.1;
53     }
54
55     while ( ! doubleQueue.isEmpty() ) {
56         doubleQueue.dequeue( dequeuedouble );
57         cout << dequeuedouble << " dequeued" << endl;
58         doubleQueue.printQueue();
59     }
60
61     return 0;
62 |

```

图 15.12 处理队列——fig15\_12.cpp

输出结果:

```

processing an integer Queue
The list is:0

```

```

The list is:0 1

```

```

The list is:0 1 2

```

```

The list is:0 1 2 3

```

```

0 dequeued
The list is:1 2 3

```

```

1 dequeued
The list is:2 3

```

```

2 dequeued
The list is:3

```

```

3 dequeued
The list is empty

```

```

processing a double Queue
The list is:1.1

```

```

The list is:1.1 2.2

```

```

The list is:1.1 2.2 3.3

The list is:1.1 2.2 3.3 4.4

1.1 dequeued
The list is:2.2 3.3 4.4

2.2 dequeued
The list is:3.3 4.4

3.3 dequeued
The list is:4.4

4.4 dequeued
The list is empty

All nodes destroyed

All nodes destroyed

```

图 15.13 图 15.12 中所示程序的输出结果

在主函数 main 中,堆栈类模板用于生成 Queue<int> 的一个实例 intQueue。整型值从 0 到 3 以先进先出的次序依次进入队列 intQueue 中然后依次出队,堆栈类模板也用于生成 Queue<double> 的一个实例 doubleQueue。双精度值 1.1,2.2,3.3,4.4 同样以先进先出的次序依次进入队列 doubleQueue 中,然后又依次出队。

## 15.7 树

链表、堆栈和队列都属于“线性数据结构”。树则是具有特殊属性的非线性的二维数据结构,每个树节点都包含两个或两个以上的链接。本节将讨论“二叉树”(如图 15.14 所示),即树中的每个节点都有包含两个链接(它们可能全空,也可能都不为空)。根节点是一棵树的第一个节点,它的每一个链接都引用了一个子节点。左边的子节点是左子树的第一个节点,右边的子节点是右子树的第一个节点。同一个节点的子节点称为兄弟节点。没有子节点的节点称为叶节点。显然,与自然界的树结构相反,计算机科学家们在绘制树时,往往从根节点开始。

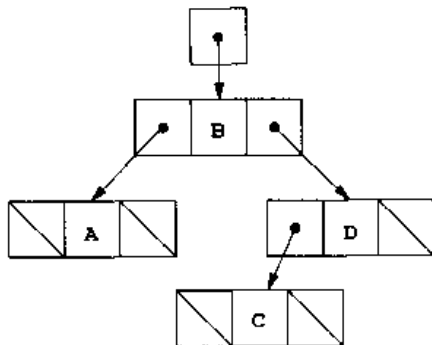


图 15.14 二叉树图示

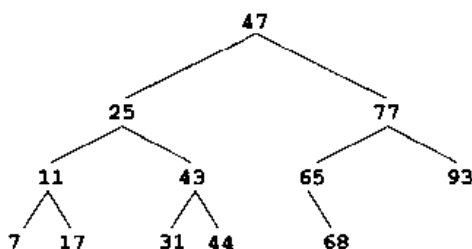


图 15.15 二叉查找树

本节要创建一种特殊的二叉树——二叉查找树。这种树上不存在重复的节点,而且它上面的任何左子树都要比其父节点小,右子树则比其父节点大。图 15.15 展示了拥有 12 个节点的二叉查找树。值得注意的是,具有相同数据的二叉查找树的形状会随着数据插入顺序的不同而不同。

**常见编程错误 15.9** 未将叶节点的链接设置为空(零)。

图 15.16(示范输出如图 15.17)的程序创建了一棵二叉查找树,并采用了 3 种遍历方式,即“前序遍历”、“中序遍历”和“后序遍历”。

```

1 //Fig.15.16: treenode.h
2 //Definition of class TreeNode
3 #ifndef TREENODE_H
4 #define TREENODE_H
5
6 template< class NODETYPE > class Tree; //forward declaration
7
8 template< class NODETYPE >
9 class TreeNode {
10     friend class Tree< NODETYPE >;
11 public:
12     TreeNode( const NODETYPE &d )
13         : leftPtr( 0 ), data( d ), rightPtr( 0 ) {}
14     NODETYPE getData() const { return data; }
15 private:
16     TreeNode< NODETYPE > *leftPtr; //pointer to left subtree
17     NODETYPE data;
18     TreeNode< NODETYPE > *rightPtr; //pointer to right subtree
19 };
20
21 #endif
  
```

图 15.16 创建并遍历二叉树——treenode.h

```

22 //Fig.15.16: tree.h
23 //Definition of template class Tree
24 #ifndef TREE_H
25 #define TREE_H
26
27 #include <iostream>
28 #include <cassert>
29 #include "treenode.h"
  
```

```
30
31 using std::endl;
32
33 template< class NODETYPE >
34 class Tree {
35 public:
36     Tree();
37     void insertNode( const NODETYPE & );
38     void preOrderTraversal() const;
39     void inOrderTraversal() const;
40     void postOrderTraversal() const;
41 private:
42     TreeNode< NODETYPE > *rootPtr;
43
44     //utility functions
45     void insertNodeHelper(
46         TreeNode< NODETYPE > **, const NODETYPE & );
47     void preOrderHelper( TreeNode< NODETYPE > * ) const;
48     void inOrderHelper( TreeNode< NODETYPE > * ) const;
49     void postOrderHelper( TreeNode< NODETYPE > * ) const;
50 };
51
52 template< class NODETYPE >
53 Tree< NODETYPE >::Tree() { rootPtr = 0; }
54
55 template< class NODETYPE >
56 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
57     { insertNodeHelper( &rootPtr, value ); }
58
59 //This function receives a pointer to a pointer so the
60 //pointer can be modified.
61 template< class NODETYPE >
62 void Tree< NODETYPE >::insertNodeHelper(
63     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
64 {
65     if ( *ptr == 0 ) { //tree is empty
66         *ptr = new TreeNode< NODETYPE >( value );
67         assert( *ptr != 0 );
68     }
69     else //tree is not empty
70         if ( value < ( *ptr )->data )
71             insertNodeHelper( &( ( *ptr )->leftPtr ), value );
72         else
73             if ( value > ( *ptr )->data )
74                 insertNodeHelper( &( ( *ptr )->rightPtr ), value );
75             else
76                 cout << value << " dup" << endl;
77 }
78
79 template< class NODETYPE >
80 void Tree< NODETYPE >::preOrderTraversal() const
```

```

81     { preOrderHelper( rootPtr ); }
82
83 template< class NODETYPE >
84 void Tree< NODETYPE >::preOrderHelper(
85     TreeNode< NODETYPE > *ptr ) const
86 {
87     if ( ptr != 0 ) {
88         cout << ptr->data << " ";
89         preOrderHelper( ptr->leftPtr );
90         preOrderHelper( ptr->rightPtr );
91     }
92 }
93
94 template< class NODETYPE >
95 void Tree< NODETYPE >::inOrderTraversal() const
96     { inOrderHelper( rootPtr ); }
97
98 template< class NODETYPE >
99 void Tree< NODETYPE >::inOrderHelper(
100     TreeNode< NODETYPE > *ptr ) const
101 {
102     if ( ptr != 0 ) {
103         inOrderHelper( ptr->leftPtr );
104         cout << ptr->data << " ";
105         inOrderHelper( ptr->rightPtr );
106     }
107 }
108
109 template< class NODETYPE >
110 void Tree< NODETYPE >::postOrderTraversal() const
111     { postOrderHelper( rootPtr ); }
112
113 template< class NODETYPE >
114 void Tree< NODETYPE >::postOrderHelper(
115     TreeNode< NODETYPE > *ptr ) const
116 {
117     if ( ptr != 0 ) {
118         postOrderHelper( ptr->leftPtr );
119         postOrderHelper( ptr->rightPtr );
120         cout << ptr->data << " ";
121     }
122 }
123
124 #endif

```

图 15.16 创建并遍历二叉树——tree.h

```

125 //Fig.15.16; fig15_16.cpp
126 //Driver to test class Tree
127 #include <iostream>
128 #include <iomanip>
129 #include "tree.h"

```

```
130
131 using std::cout;
132 using std::cin;
133 using std::setiosflags;
134 using std::ios;
135 using std::setprecision;
136
137 int main()
138 {
139     Tree< int > intTree;
140     int intVal, i;
141
142     cout << "Enter 10 integer values:\n";
143     for( i = 0; i < 10; i++ ) {
144         cin >> intVal;
145         intTree.insertNode( intVal );
146     }
147
148     cout << "\nPreorder traversal\n";
149     intTree.preOrderTraversal();
150
151     cout << "\nInorder traversal\n";
152     intTree.inOrderTraversal();
153
154     cout << "\nPostorder traversal\n";
155     intTree.postOrderTraversal();
156
157     Tree< double > doubleTree;
158     double doubleVal;
159
160     cout << "\n\nEnter 10 double values:\n"
161         << setiosflags( ios::fixed | ios::showpoint )
162         << setprecision( 1 );
163     for ( i = 0; i < 10; i++ ) {
164         cin >> doubleVal;
165         doubleTree.insertNode( doubleVal );
166     }
167
168     cout << "\nPreorder traversal\n";
169     doubleTree.preOrderTraversal();
170
171     cout << "\nInorder traversal\n";
172     doubleTree.inOrderTraversal();
173
174     cout << "\nPostorder traversal\n";
175     doubleTree.postOrderTraversal();
176
177     return 0;
178 }
```

图 15.16 创建并遍历二叉树——fig15\_16.cpp

输出结果:

```
Enter 10 integer values;
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50

Enter 10 double values;
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 89.5 92.5 90.8 82.7 39.2
```

图 15.17 图 15.16 中所示程序的输出结果

main 函数首先定义了一个 `Tree <int>` 类型的整数型二叉树 `intTree`, 然后程序提示输入 10 个整型数据, 随后调用函数 `insertNode` 将它们插入二叉树, 之后程序对 `intTree` 进行前序遍历、中序遍历和后序遍历。随后的程序定义了一个 `Tree <double>` 类型的浮点二叉树 `doubleTree`, 程序提示输入 10 个浮点数据, 然后调用 `insertNode` 将它们插入二叉树, 最后程序对 `doubleTree` 进行前序遍历、中序遍历和后序遍历。

现在我们讨论有关类模板的定义。类模板 `TreeNode` 将类模板 `Tree` 声明为友元类, 然后将节点数据 `data`、指针 `leftPtr` (指向左子树) 和 `rightPtr` (指向右子树) 声明为 `private` 成员。构造函数则将 `data` 设为传递过来的参数值, 并将 `leftPtr` 与 `rightPtr` 设置为空 (即使它成为叶节点)。成员函数 `getData` 用于返回 `data` 的值。

`Tree` 类拥有 `private` 指针成员 `rootPtr` 指向树的根节点, 同时它也拥有 `public` 成员函数 `insertNode` 和 `preorderTraversal`, `inorderTraversal`, `postorderTraversal`。这些成员函数都调用了相对独立的递归函数来执行相应操作。其构造函数则将 `rootPtr` 初始化为 0 以表明树的初始状态为空。

`Tree` 类的功能函数 `insertNodeHelper` 采用递归的方式将节点插入树, 节点在二叉查找树中只能以叶节点的形式插入。如果树为空, 则创建一个新的树节点 `TreeNode`, 初始化并将其插入树。

如果不为空, 程序就比较插入值与根节点的大小。如果插入值较小, 程序则调用 `insertNodeHelper` 在左子树中插入该节点, 如果插入值比根节点大, 程序就将其插入右子树。如果插入值与根节点大小相等, 程序则不进行插入操作并输出“重复”消息并返回。

3 个成员函数 `preorderTraversal` 和 `inorderTraversal` 和 `postorderTraversal` 都对树进行遍历, 并输出树节点数据。

中序遍历 `inorderTraversal` 的步骤如下:

(1) 用 `inorderTraversal` 遍历左子树;



(2) 处理节点中的值(也就是打印节点值);

(3) 用 `inorderTraversal` 遍历右子树。

节点值只有其左子树节点中的值处理完之后才能得以处理。图 15.18 所示树的中序遍历结果为

6 13 17 27 33 42 48

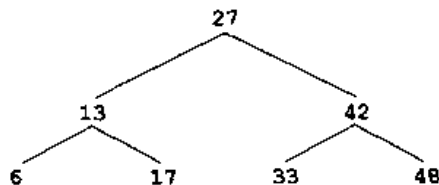


图 15.18 二叉查找树

注意,二叉查找树的中序遍历 `inorderTraversal` 是以递增的次序来处理节点值的。创建二叉查找树的过程实际上是对树节点值进行排序的过程,因此这个过程也称为二叉树排序。

前序遍历 `PreorderTraversal` 的步骤如下:

(1) 处理节点中的值(也就是打印节点值);

(2) 用 `PreorderTraversal` 遍历左子树;

(3) 用 `PreorderTraversal` 遍历右子树。

访问了指定节点之后,才可以访问其左子树,最后是其右子树。图 15.18 所示树的前序遍历结果为:

27 13 6 17 42 33 48

后序遍历 `PostorderTraversal` 的步骤如下:

(1) 用 `PostorderTraversal` 遍历左子树;

(2) 用 `PostorderTraversal` 遍历右子树;

(3) 处理节点中的值(也就是打印节点值)。

对于指定节点,只有在其所有子节点已访问这后才能得以访问。图 15.18 所示树的后序遍历结果为

6 17 13 33 48 42 27

二叉查找树使消除重复节点变得很容易。创建树的过程中,重复节点按照它与其他节点的比较结果进入左子树或右子树,最后它会与其具有相同值的节点进行比较。如此一来,重复节点便很容易的识别,并删除。

按关键字在二叉查找树查找值的速度很快。如果树是平衡树,即每一层的节点数是上一层的两倍,此时一棵拥有  $n$  个节点的二叉树最多有  $\log_2 n$  层。因些要查找一个值是否存在最多只需要进行  $\log_2 n$  次比较。这就意味着在一棵有 1 000 个节点的二叉查找树中,寻找一个值所需的比较次数不会超过 10 次( $2^{10} > 1\,000$ ),而在一棵拥有 1 000 000 个节点的二叉查找树中查找一个值所需的比较次数不超过 20 次( $2^{20} > 1\,000\,000$ )。

本章练习题也提供了对二叉查找树进行其他操作的算法如从二叉查找树中删除节点,以二维的格式打印其中的数据并按层进行遍历。二叉查找树的按层遍历就是先访问根节点层,然后逐层地向下访问,在每一层中,从左至右被访问节点。其他有关二叉查找树的练习包括允许二叉查找树包含重复节点,将字符值插入树以及确定指定二叉查找树的层数。

## 15.8 小结

- 自引用类包含名为链接的成员,链接是指向具有相同类类型对象的指针。
- 自引用类使得多个对象可以以堆栈、队列和树的形式链接起来。
- 动态内存分配在程序执行时开辟一块内存空间以存储新的对象。
- 链表是自引用类对象的线性集合。
- 链表是一种动态的数据结构,其长度可随需要增加或减小。
- 链表可以持续增长直到内存消耗殆尽。
- 链表提供了一种机制可通过指针操作来插入和删除对象的。
- 单向链表以指向第一个节点的指针开始,每个节点都包含了指向序列中下一个节点的指针,同时以一个指针成员为空的节点结束。单向链表只能单遍历。
- 循环单向链表以指向第一个节点的指针开始,然后每一个节点都包含一个指向链表中下一节点的指针,但不同的是它的最后一个节点的指针成员不为空,而是指向第一个节点,因此称为循环单向链表。
- 双向链表允许前后两个方向的链表遍历。这种链表通常有两个起始指针,一个指向第一个节点以用于由前向后的链表遍历,另一个指向最后一个节点以用于由后向前的链表遍历。
- 在循环双向链表中,最后一个节点向后的指针指向第一个节点,而第一个节点向前的指针则指向最后一个节点。
- 堆栈与队列都是增添了某些限制的链表。
- 堆栈节点只能在栈顶增加与删除。
- 堆栈中最后一个节点的指针成员应该被置为空以表明到达栈底。
- 堆栈的两种主要操作是压入堆栈 push 与弹出堆栈 pop。操作 push 创建一个新节点并将其置于栈顶,操作 pop 从栈顶删除一个节点,释放其所占用的内存空间并返回节点值。
- 在队列中,节点在尾部增加而在头部删除,因此队列被称为先进先出(FIFO)的数据结构。增加与删除节点操作分别为 enqueue 和 dequeue。
- 树是一种二维的数据结构,每个节点有两个或两个以上的指针成员。
- 二叉树每个节点包含两个链接。
- 根节点是树的第一个节点。
- 树的每一个链接都指向它的一个子节点。左边的子节点即左子树的根节点,右边的子节点即右子树的根节点。同一个节点的子节点称为兄弟节点。没有子节点的节点被称为叶节点。
- 二叉查找树上左子树上的所有值都比其父节点小,而右子树上的所有值则比其父节点大或相等。如果没有重复的节点值,则右子节点中的值总是大于其父节点中的值。
- 中序遍历先处理左子树,再处理根节点,最后再处理右子树。一个节点只有当它的

左子树被访问完毕后才能被访问。

- 前序遍历先处理根节点,再依次处理左子树与右子树。遇到节点立即进行处理。
- 后序遍历先依次处理左子树与右子树,再处理根节点。对节点而言,只有其子节点处理完之后,才能得以处理。

## 本章术语

binary search tree 二叉查找树

binary tree sort 二叉树排序

binary tree 二叉树

child node 子节点

children 子子树

circular, doubly linked list 循环双向链表

circular, singly linked list 循环单向链表

deleting a node 删除节点

dequeue 出队

double indirection 双间接

doubly linked list 双向链表

duplicate elimination 重复消除

dynamic data structure 动态数据结构

dynamic memory allocation 动态内存分配

enqueue 入队

FIFO(first-in, first-out) 先进先出

head of a queue 队列头

inorder traversal of a binary tree 二叉树中序遍历

inserting a node 插入节点

leaf node 叶节点

left child 左节点

left subtree 左子树

level-order traversal of a binary tree 二叉树的层序遍历

LIFO(last-in, first-out) 后进先出

linked data structure 线性数据结构

linkod list 链表

node 节点

nonlinear data structure 非线性数据结构

null pointer 空指针

parent node 父节点

pointer to a pointer 指向指针的指针

pop 弹出

postorder traversal of a binary tree

二叉树后序遍历

predicate function 判断函数

preorder traversal of a binary tree 二叉树前序遍历

push 压入栈

queue 队列

right child 右节点

right subtree 右子树

root node 根节点

self-referential structure 自引用类结构

siblings 兄弟节点

singly linked list 单向链表

sizeof 操作符

stack 堆栈

subtree 子树

tail of a queue 队列尾

top 顶部

traversal 遍历

tree 树

visit a node 访问一个节点

## 常见编程错误

15.1 未将链表最后一个节点的指针设为空(零)。

15.2 认为类对象的长度等于其数据成员长度的总和。

15.3 没有释放不再需要的动态分配内存会导致系统过早耗尽内存,这有时称为内存泄漏。

15.4 用操作符 delete 释放不是用操作符 new 动态分配的内存。

15.5 引用已被删除的内存。

15.6 试图删除已被删除的内存可能会在运行时造成不可预料的结果。

15.7 未将栈底节点的指针设为空(零)。

15.8 未将队列中最后一个节点的指针设置为空(零)。

15.9 未将叶节点的指针设置为空(零)。

### 良好编程习惯

15.1 不再需要操作符 new 分配的内存时,立刻用操作符 delete 释放内存

15.2 把空值(零)赋值给新节点的成员,使用指针前应进行初始化。

### 性能提示

15.1 在声明时,可以把数组元素的个数定义为多于数据项个数,但这样会浪费内存。此时,链表能够更有效地利用内存,它允许程序在运行时进行调整。

15.2 数组的插入与删除操作可能很耗时间,因为插入点和删除点之后的所有元素都要作相应移动。

15.3 数组元素在内存中是连续存放的,这样一来,就可直接存取数组元素,因为任何数组元素的地址都可以直接通过它与数组中第一个元素的相对位置来确定。而链表不能直接存取其中的元素。

15.4 如用动态内存分配(而不是数组)来处理对于那些运行时会动态增长或缩小的数据结构,可节约存储空间。但要记住,指针也会占用空间,而且动态内存分配会引起调用函数的开销。

### 可移植性提示

15.1 类对象的大小不一定是其数据成员大小的总和,这是因为有的机器要求存储内容必须沿存储单元的边界开始存储,当然还有别的原因(参见第16章)。因此尽量用操作符 sizeof 来决定对象的大小。

### 自测题

15.1 填空题:

- a) \_\_\_\_\_类用于生成程序执行时能动态增长和缩小的数据结构。
- b) 操作符\_\_\_\_\_用于动态分配内存和构造对象,并且会返回对象的指针。
- c) \_\_\_\_\_是链表的增加了限制的变体,它的增加、删除操作只能在起始处进行,节点值以后进先出的次序返回。
- d) 不改变链表,只是确定链表是否为空的函数称为\_\_\_\_\_。
- e) 队列因为其第一个增加的节点是最先离开的节点而被称为\_\_\_\_\_。
- f) 在链表中指向下一个节点的指针称为\_\_\_\_\_。
- g) 操作符\_\_\_\_\_被用来删除对象并释放对象所占用的内存空间。
- h) \_\_\_\_\_是链表的增加了限制的变体,它的节点增加操作只能在尾部进行,删除操作只能在头部进行。
- i) \_\_\_\_\_是一种非线性的二维数据结构,其节点包含两个或两个以上的指针。
- j) 堆栈因为其最后进堆栈的节点最先离堆栈而被称为\_\_\_\_\_的数据结构。

- k) \_\_\_\_\_树的节点只包含两个指针成员。  
 l) 树的第一个节点是\_\_\_\_\_。  
 m) 树节点的每一个指针都指向这个节点的\_\_\_\_\_或\_\_\_\_\_。  
 n) 没有子节点的树节点被称为\_\_\_\_\_。  
 o) 本章介绍的二叉查找树的4种遍历算法分别为\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。

15.2 链表与堆栈区别何在?

15.3 堆栈与队列区别何在?

15.4 更适合本章内容的标题可能是“可重用的数据结构”。评价以下概念在数据结构重用中的作用。

- a) 类  
 b) 类模板  
 c) 继承  
 d) private 继承  
 e) 合成

15.5 手工求出图 15.19 中二叉查找树的中序、前序、后序遍历结果。

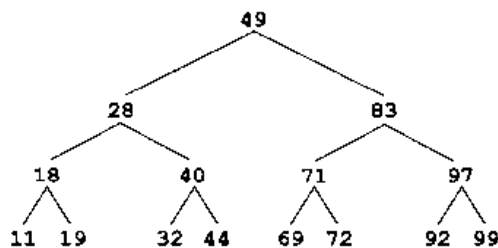


图 15.19 包含 15 个节点的二叉查找树

### 自测题答案

- 15.1 a) 自引用类 b) new c) 堆栈 d) 判断函数 e) 先进先出(FIFO) f) 链接  
 g) delete h) 队列 i) 树 j) 后进先出(LIFO) k) 二叉树 l) 根节点 m) 子节点或子树 n) 叶节点 o) 中序,前序,后序
- 15.2 可在链表的任何位置插入或删除节点,而堆栈只能在栈顶插入或删除节点。
- 15.3 队列拥有指向其头部和尾部的指针,因此插入节点可在尾部进行,删除节点在头部进行。而堆栈只有一个指向栈顶的指针,增加和删除节点都在栈顶进行。
- 15.4 a) 类允许我们构造任意多的某一类型的数据结构对象(即类的实例)  
 b) 类模板能够实例化相关的一些类,其中每个类都基于不同类型的参数。同时也能构造一个类模板的任意多的实例。  
 c) 继承使我们可以在派生类中重用基类的代码,因而派生类具有与基类相同的数据结构(采用 public 方式继承时)。  
 d) private 继承方式使我们可以在派生类中重用基类的部分代码以便形成派生类自己的数据结

构。因为继承是 `private` 的,基类中所有的 `public` 函数变为 `private`,这就防止了派生类中数据结构的使用者能够访问在派生类中并未生效的基类函数。

- e) 合成可以让类对象数据结构变成合成类的成员,从而使得代码可以重用;如果我们把这个类对象变为合成类的 `private` 成员,那么该类对象的 `public` 成员函数就无法通过合成对象的接口进行访问。

15.5 中序遍历结果为:

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

前序遍历结果为:

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

后序遍历结果为:

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

## 练习题

15.6 编写程序将两个字符型链表对象链接起来。程序应该包括函数 `concatenate`,它取两个链表对象的引用作为参数,并将第二个链表对象与第一个链接在一起。

15.7 编写程序将两个有序的链表对象合并为一个有序的链表对象。程序应该包括函数 `merge`,它以两个链表对象的引用作为参数,并返回合并后的链表对象。

15.8 编写程序将 25 个在 0 到 100 之间的任意整数插入到一个有序的链表中。程序应计算出它们的总和并求出它们的浮点平均值。

15.9 编写程序创建一个包含有 10 个字符的链表同时,也创建另外一个仅字符顺序相反的链表。

15.10 编写程序输入一行文本然后利用堆栈对象将它们逆序输出。

15.11 编写程序使用堆栈对象以确定一个字符串是否为回文(指顺读和倒读都一样的词语)。程序应该忽略空格与标点。

15.12 堆栈在编译器中用于计算表达式和产生机器代码。在本练习题和下一个练习题里,我们要研究编译器如何计算只包括常量、操作符和括号的表达式。

人们书写表达式时,通常习惯把操作符写在两个操作数的中间,这称为中缀表示法,如  $3 + 4$  和  $7 / 9$ 。计算机却习惯于把操作符写在两个操作数的后面,这被称为后缀表示法。例如,前面两个表达式的后缀表示分别为  $14 +$  和  $79 /$ 。

要计算复杂的中缀表达式,计算机首先要将它转换为后缀表达式,然后再进行计算。转换及计算的算法都是对表达式进行从左至右的扫描。在这两种算法中都要用到堆栈,但堆栈的作用却不相同。

这个练习题要求你写一个 C++ 程序以实现中缀表达式转换为后缀表达式的算法。下一个练习则要求你编写 C++ 程序实现计算后缀表达式的算法。通过后文的描述,你会发现这两个练习题中所写的程序将有助于实现一个完整的可工作的编译器。

编写程序,把只含一位整数的普通中缀代数表达式(假设输入的表达式是有效的)例如

$6 + 2) * 5 - 8 / 4$

转换为后缀表达式即

6 2 + 5 \* 8 4 / -

程序首先将表达式读入一个字符数组,然后用修改过的堆栈函数在字符数组 postfix 中创建后缀表达式。算法如下:

- (1) 将左括号压入堆栈中;
- (2) 在数组 infix 中增加一个右括号;
- (3) 堆栈非空时,从左至右扫描数组 infix,并根据当前扫描的字符,执行如下工作:
  - 如果当前字符是一个数字,则将其复制到数组 postfix 中;
  - 如果当前字符是左括号,则将其压入堆栈中;
  - 如果当前字符是操作符,则如果栈顶存在操作符并且优先级与当前操作符相等或比当前操作符高,则将其弹出堆栈且插入到数组 postfix 中并将当前字符压入到堆栈中;
  - 如果当前字符为右括号,则从堆栈中弹出操作符并将它们插入到数组 postfix 中直到遇到左括号。然后弹出(放弃)堆栈中的左括号。

下列操作符可用于表达式

- + (加号)
- (减号)
- \* (乘号)
- / (除号)
- ^ (乘方)
- % (求模)

堆栈里的每个节点应该包含一个数据成员和一个指向下一个堆栈节点的指针。

可能要编写的函数为:

- a) 函数 convertToPostfix 将中缀表达式转换为后缀表达式;
- b) 函数 isOperator 确定一个字符是否为操作符;
- c) 函数 precedence 决定两个操作符之间的优先级。按照大于,等于,小于的关系分别返回 -1,0,1;
- d) 函数 push 将一个值压入堆栈中;
- e) 函数 pop 从堆栈中弹出一个值;
- f) 函数 stackTop 返回栈顶元素的值但不弹出它;
- g) 函数 isEmpty 确定堆栈是否为空;
- h) 函数 printStack 打印堆栈中数据。

15.13 编写程序计算后缀表达式的值如 6 2 + 5 \* 8 4 / -。程序首先将包含数字与操作符的后缀表达式读入到一个字符数组中,然后使用修改过的本章前面所述的堆栈函数来扫描表达式并进行计算。算法如下:

- (1) 在后缀表达式数组的末端增加空字符(' \0' ),以便程序在遇到空字符即停止处理。
- (2) 当没遇到空字符时,从左至右扫描表达式。如果扫描到的当前字符是数字,则将其压入堆栈中(一个数字字符的整数值为其 ASCII 值减去 '0' 的 ASCII 值)。如果

当前字符为操作符,则弹出栈顶的两个元素并保存到变量  $x$  和  $y$  中,然后计算  $x$  (操作符)  $y$  的值并将运算结果压入堆栈中。

(3) 当遇到空字符时,弹出栈顶值即为所求结果。

注意:在步骤2)中,如果操作符为“/”,栈顶元素为2,其下一个元素为8,则将2保存到变量  $x$  中,8保存到变量  $y$  中,计算  $8/2$ ,将计算结果4压入堆栈中。其他操作符类同。在表达式中允许使用的操作符有: + (加号), - (减号), \* (乘号), / (除号), ^ (乘方), % (求模)。堆栈中节点应该包含一个 `int` 类型的数据成员和一个指向下一个节点的指针。需要编写的函数有:

- a) 函数 `evaluatePostfixExpression`, 用于计算后缀表达式;
- b) 函数 `calculate` 用于计算;
- c) 函数 `push` 用于把值压入堆栈;
- d) 函数 `pop` 用于弹出栈顶值;
- e) 函数 `isEmpty` 用于确定堆栈是否为空;
- f) 函数 `printStack` 用于打印堆栈中数据。

15.14 修改 15.13 中的程序,使之能处理大于9的整型数据。

15.15 (超级市场模拟程序)编写程序模拟超级市场中等候付款的队列,这可看成队列对象。每隔1~4(1~4之间的整数)分钟有一个顾客(看成顾客对象)进入队伍(即入队),同时,每隔1~4(1~4之间的整数)分钟有一个顾客获得服务(即出队)。显然,入队率与出队率之间要进行平衡,否则如果入队率大于出队率,队列将会无限制增长。即使在入队率与出队率平衡的情况下,由于出队与入队的随机性队列也可能变得很长。用如下算法来运行超市模拟程序,持续时间一天(以12小时计算):

- (1) 选择1~4之间的一个整数以确定第一个顾客入队的时间
- (2) 当第一个顾客入列时,确定他等待服务的时间(1~4之间的随机整数);然后开始为顾客服务;最后确定下一个顾客到来的时间(也是1~4之间的随机整数)
- (3) 在这一天每一分钟,如果下一个顾客到,则输出顾客到的消息并使之入队同时计划好再下一个顾客到来的时间。如果有一个顾客已获得服务,则输出服务完毕的消息并令下一个顾客出队以接受服务并确定完成服务所需的时间。

运行该模拟程序一天(以12小时计算),并回答下列问题:

- a) 在任何时刻,队列中最大顾客数为多少?
- b) 一个顾客的最长等待时间?
- c) 如果将以上的时间间隔从1~4分钟改为1~3分钟会出现什么情况?

15.16 修改图 15.16 中的程序以允许二叉树中包含重复值。

15.17 编写基于图 15.16 的程序:输入一行文本,将其分解为单个的单词(可以使用库函数 `strtok`),然后将它们逐个插入二叉树并打印出中序、前序、后序的遍历结果。注意用面向对象(OOP)的方法。

15.18 本章,你可以看到用创建二叉树来消除重复值非常简单。试描述如何用一个一维数组来消除重复值,并比较这两种方法性能上的差异。

15.19 写一个函数确定一棵树包含的层数。



- 15.20 (以逆序递归打印链表)写链表的一个成员函数 `printListBackwards` 以逆序递归输出链表中的数据。同时写一个测试程序创建一个链表并调用该函数逆向打印链表中的数据。
- 15.21 (递归查找链表)写链表的一个成员函数 `searchList` 在链表中递归查找指定值。如果找到指定值,则返回指向它的指针,否则返回空。再写一个测试程序创建一个链表并调用此函数。程序应提示用户输入要查找的值。
- 15.22 (删除二叉树节点)在这个练习里,我们来讨论如何删除二叉查找树上的一个节点。其算法不如插入节点那么简单明了。这里有三种情况需要考虑:待删节点为叶节点、待删节点有一个子节点以及待删节点有两个子节点。

如果待删节点为叶节点,直接删除此节点并将其父节点中相应的指针设置为空即可。如果待删节点有一个子节点,则将其父节点中的相应指针指向其子节点再将其删除即可。也即待删节点的子节点取代了待删节点。

第三种情况最为困难,此时待删节点有两个子节点。当它被删除时,必须要有另外一个节点来取代它。然而其父节点中相应的指针不能指向它的任意一个子节点。在大多数情况下,二叉查找树(无重复值)的下列特性会受到破坏:其上任意一个左节点的值要比其父节点小,而右节点的值则比其父节点大。

那么哪一个节点应该来取代待删节点的位置而又能保持二叉查找树的特性呢?答案为:要么是小于待删节点值的节点中最大值,或者是大于待删节点值的节点中最小值。在二叉查找树中,小于父节点值的节点中具有最大值节点位于父节点的左子树中且是最靠右的一个。这个节点能够通过从左子树中向右查找直到一个节点的指向右子树的指针为空为止。我们找到的这个节点(在这里我们称为取代节点)要么是一个叶节点要么只有一个左子树。如果取代节点是一个叶节点,就可采用以下步骤删除节点:

- (1) 保存待删节点的指针到一个临时变量(此指针用于释放动态分配的内存);
- (2) 将待删节点的父节点中相应的指针指向取代节点;
- (3) 将取代节点的父节点中相应的指针置为空;
- (4) 将取代节点指向右子树的指针指向待删节点的右子树;
- (5) 删除临时变量所指的节点(即待删节点)。

如果取代节点有一个左子节点,其算法与以上类似,但必须要将其左子节点移至取代节点的位置。删除节点步骤如下:

- (1) 保存待删节点的指针到一个临时变量;
- (2) 将待删节点的父节点中相应的指针指向取代节点;
- (3) 将取代节点的父节点中相应的指针指向其左子树;
- (4) 将取代节点指向右子树的指针指向待删节点的右子树;
- (5) 删除临时变量所指的节点(即待删节点)。

编写成员函数 `deleteNode`,其参数为指向所要操作的二叉树根节点的指针和待删除节点值。程序要定位待删节点并且利用上面所述算法删除之。如果找不到所要删除的值,则打印消息提示删除不成功。修改图 15.16 中的程序来使用这个函数。删除节

点后,调用遍历函数 `inOrder`, `preOrder` 和 `postOrder` 来确认删除操作被正确进行。

15.23 (二叉树查找)编写成员函数 `binaryTreeSearch` 来定位二叉树中的特定值。函数参数为指向所要操作的二叉树根节点的指针和要定位的节点值。如果定位成功,则返回指向所定位节点的指针,否则返回空指针。

15.24 (层序遍历二叉树)图 15.16 的程序演示了前序、中序和后序遍历二叉树。本练习要进行二叉树的层序遍历,即从根节点层开始,按层打印二叉树的节点值。其算法不是递归算法,它用队列控制节点值的输出。算法如下

(1) 将根节点插入到队列中

(2) 当队列中还有节点时,获取队列中的下一个节点并打印其值。如果此节点的左子树非空,则将其左子树插入到队列中。如果节点的右子树非空,也将其右子树插入到队列中。

编写成员函数 `levelOrder` 以执行层序遍历,修改图 15.16 中的程序来使用这个函数(同时也需要修改图 15.12 中的队列处理程序)。

15.25 (打印树)编写一个递归函数 `outputTree` 在屏幕上显示所有节点。函数按层输出节点,树的最顶层在屏幕的最左边,最底层在屏幕的最右边。显示输出是垂直进行的。例如,图 15.19 中的二叉树输出为

```

          99
        97
       92
      83
     72
    71
   69
  49
 44
40
32
28
19
18
11

```

注意叶节点屏幕的最右列,而根节点在最左列,列与列之间用 5 个空格分隔。函数 `outputTree` 应该带有一个参数 `totalSpaces` 以表示输出一个值前所应输出的空格数(此参数以 0 开始表示根节点在屏幕最左边)。函数使用经修改了的中序遍历函数来输出树节点,它先输出最右边的节点,然后再从右至左输出节点值。算法如下:

在当前节点的指针不为空的情况下,取参考数 `totalSpaces + 5`,递归调用函数 `outputTree` 输出当前节点的右子树;用 `for` 循环结构计算并输出 1 到 `totalSpaces` 之间的空格;输出当前节点值;把指针设为指向前节点的左子树;`totalSpaces` 自增 5。

### 特色部分:构建自己的编译器

在练习题 5.18 与练习题 5.19 中,我们介绍了 Simpleton 机器语言(SML)并实现了用 Simple-

ton 模拟器来执行 SML 程序。在本节中,我们要构建一个编译器,将一个高级语言程序转换为 SML。除了讲述如何构建编译器,本部分还要讨论如何用这种高级语言编程。你首先要用这种新的高级语言来编写程序,然后在自己构建的编译器上进行编译,最后在练习题 7.19 实现的模拟程序上运行。应该尽量用面向对象的方法来构建这个编译器。

15.26 (Simple 语言)在开始构建编译器之前,我们来讨论一种与流行的 BASIC 语言的早期版本类似的虽然简单但功能强大的高级语言,我们称之为 Simple 语言。每一条 Simple 语句包括一个行号和一条 Simple 指令。其中行号必须以递增的顺序出现。每一条指令以下面的 Simple 命令开始:rem, input, let, print, goto, if / goto 和 end(参见图 15.20)。除了 end 之外的所有命令都可以重复使用。Simple 只能计算使用加(+)、减(-)、乘(\*)、除(/)这 4 个操作符的表达式。这些操作符的优先级与 C 语言中的操作符优先见相同。当然,也可用括号来改变计算顺序。

| 命令      | 示范语句                    | 说明                                            |
|---------|-------------------------|-----------------------------------------------|
| rem     | 50 rem this is a remark | rem 之后出现的文本用于文档记录,将被编译器忽略                     |
| input   | 30 input x              | 显示一个问号,提醒用户输入一个整数。读取通过键盘输入的整数并将该整数保存在 x 中     |
| let     | 80 let u = 4 * (j - 56) | 将 4 * (j - 56) 的值赋给 u。注意等号右边可能出现较为复杂的表达式      |
| print   | 10 print w              | 显示 w 的值                                       |
| goto    | 70 goto 45              | 将程序控制转到第 45 行                                 |
| if/goto | 35 if i == z goto 80    | 比较 i 与 z 是否相等。如果条件为真则程序控制转到第 80 行,否则继续执行下一条语句 |
| end     | 99 end                  | 终止程序执行                                        |

图 15.20 Simple 命令

我们的 Simple 编译器只能识别小写字母。在一个 Simple 文件里的所有字符都应该都是小写的(大写字母会导致语法错误,除非它们出现在 rem 语句中,而在 rem 语句中大小写是被忽略的)。变量名是单个的字母,而且 Simple 语言不允许描述性的变量命名。因此,变量应该在注释行中进行注释以表明其用途。Simple 没有变量声明,而仅仅是在使用变量的时候就对它进行声明并自动将它初始化为零。Simple 的语法中并不包括字符串操作(读、写并比较字符串等)。如果在 Simple 程序中碰到一个字符串(出现在非 rem 行中),编译器就会产生语法错误。我们的编译器的最初版本假定程序输入是正确的。练习题 15.29 要求你修改编译器以执行语法错误检查。

Simple 使用条件语句 if/goto 和无条件语句 goto 来改变程序执行时的控制流。如果 if/goto 语句中的条件为真,程序控制就会被转到指定的行。在 if/goto 语句中可以使用如下的关系操作符: <, >, <=, >=, == 和 !=。这些操作符的优先级与 C++ 里相同。

现在让我们看几个程序来演示 Simple 语言的特性。第一个程序(图 15.21)从键盘接收两个整数输入并将保存到变量 a 和 b 中,最后计算并打印它们的和。

```

1 10 rem  determine and print the sum of two integers
2 15 rem
3 20 rem  input the two integers
4 30 input a
5 40 input b
6 45 rem
7 50 rem  add integers and store result in c
8 60 let c=a+b
9 65 rem
10 70 rem  print the result
11 80 print c
12 90 rem terminate program execution
13 99 end

```

图 15.21 用于计算两个整数之和的 Simple 程序

图 15.22 中的程序确定并打印两个整数中的较大值。两个整数从键盘输入并保存到变量  $s$  和  $t$  中,然后 `if/goto` 语句测试条件  $s \geq t$ ,如果条件为真,控制就转到第 90 行同时输出变量  $s$  的值,否则输出变量  $t$  的值,然后程序控制转到第 99 行中的 `end` 语句,程序即告终止。

```

1 10 rem  determine the larger of two integers
2 20 input s
3 30 input t
4 32 rem
5 35 rem  test if s >=t
6 40 if s >=t goto 90
7 45 rem
8 50 rem  t is greater than s,so print t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem  s is greater than or equal to t,so print s
13 90 print s
14 99 end

```

图 15.22 用于查找两个整数中较大值的 Simple 程序

Simple 没有提供循环结构(如 C++ 里面的 `for`, `while` 和 `do/while` 循环)。然而,Simple 能够用 `if/goto` 和 `goto` 语句模拟 C++ 的每一种循环结构。图 15.23 中的程序使用一个监视值控制的循环来计算几个整数的平方。每一个整数从键盘输入并储存到变量  $j$  中。如果输入的值为监视值  $-9999$ ,程序控制就转向第 99 行,程序即终止。否则将  $j$  的平方赋给  $k$ ,然后将  $k$  的值输出到屏幕,程序控制转到第 20 行,等待输入下一个整数并进行处理。

```

1 10 rem  determine and print the sum of two integers
2 20 input j
3 23 rem
4 25 rem  test for sentinel value
5 30 if j == -9999 goto 99
6 33 rem

```

```

7 35 rem calculate square of j and assign result to k
8 40 let k = j * j
9 50 print k
10 53 rem
11 55 rem loop to get next j
12 60 goto 20
13 99 end

```

图 15.23 计算多个整数的平方

参考图 15.21, 15.22, 15.23 中的程序, 编写 Simple 程序完成以下任务:

- a) 输入 3 个整数, 计算它们的平均值并打印结果;
- b) 使用一个监视值控制的循环输入 10 个整数, 打印并计算它们的和;
- c) 使用计数循环输入 7 个整数, 包括正整数和负整数, 然后计算并打印它们的平均值;
- d) 输入一些整数, 确定并打印它们的最大值。输入的第一个整数为要处理的整数的数目;
- e) 输入 10 个整数打印它们的最小值;
- f) 计算并打印从 2 ~ 30 中的偶数的和;
- g) 计算并打印从 1 ~ 9 中的奇数的乘积。

15.27 (构建一个编译器; 前提: 完成练习题 5.18, 5.19, 5.12, 5.13 和 5.26) 练习题 15.26 中, 我们已经描述了 Simple 语言, 现在来构建一个 Simple 编译器。首先, 看 Simple 程序是如何转换为 SML 并在 Simpletron 模拟器上执行的 (参见图 15.24): 编译器从文件读入 Simple 程序并将其转换为 SML, 然后将 SML 输出到一个文件 (文件的每一行为一条 SML 指令), 最后 Simpletron 加载并开始执行 SML 文件。计算结果同时送往磁盘上的文件与屏幕。注意练习 5.19 中的 Simpletron 是从键盘获得输入的, 因此必须修改它以便它能从文件读入数据以便与我们的编译器保持一致。

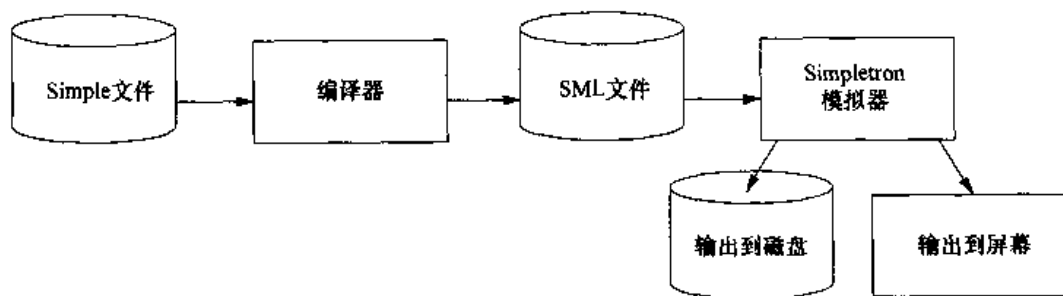


图 15.24 编写、编译并执行一个 Simple 语言程序

Simple 编译器将在 Simple 程序转换为 SML 的过程中进行两次编译。第一次编译要构建一个字符表 (对象), 字符表里保存了每一个行号 (对象)、变量 (对象)、常量 (对象), 它们的类型以及在最终的 SML 代码文件中相应的位置 (稍后会详细讨论字符表)。第一次编译也要为每一条 Simple 语句 (对象) 产生相应的 SML 指令 (对象)。

我们可以看到如果 Simple 程序包含了将程序控制权转到其后语句的语句,那么第一次编译将会在 SML 程序中产生一些“不完整”的指令。第二次编译将定位并完成这些不完整的指令,并将 SML 程序输出到一个文件。

### 第一次编译

编译器首先将一条 Simple 程序的语句读入到内存。这一行语句必须分解为单个的标识符,以便处理和编译(标准库函数 strtok 可以执行这一工作)。记住,每一行语句都以行号开始,后面再跟一条命令。当编译器将一行语句分解为单个的标识符时,如果标识符是一个行号、变量或常量,就会被放入字符表。只有一个行号是一条语句中的第一个标识符时,它才会放入字符表。symbolTable 对象是一个 tableEntry 对象的数组,每个 tableEntry 代表程序中的标识符。程序中标识符的个数目是没有限制的。因此,某些程序的字符表 symbolTable 可能会很大。现在我们将 symbolTable 设为包含 100 个元素的数组。一旦程序开始工作,还可以增加或减小它的大小。

每一个 tableEntry 对象都包括 3 个成员:成员 symbol 是一个整型值,代表着一个变量、行号或者常量的 ASCII 值;成员 type 只能取下列几个字符:“C”代表常量;“L”代表行号;“V”代表变量;成员 location 包含了标识符在 Simpletron 内存中的位置(从 00 到 99)。Simpletron 内存是拥有 100 个整型数据的数组,里面存放了 SML 指令和数据。对行号来说,location 值为其所在的 simple 语句的第一条指令(注意,一条 simple 语句可编译成多条指令)在 Simpletron 内存中的位置。对变量和常量来说,location 值是它在 Simpletron 内存中存放的位置。变量和常量在 Simpletron 内存中是按向后的次序存放的。例如,第一个变量或者常量存放在位置 99,下一个变量或常量就存放在位置 98,如此类推。

在将 symbol 程序转换为 SML 的过程中,字符表必不可少。在第 5 章中我们已经说过 SML 指令是一个四位数的整数,它包括两部分:前一部分表示操作码,后一部分表示操作数。操作码是由 Simple 里的命令所决定的。例如,symbol 命令 input 对应于 SML 操作码 10(代表“读数据”),symbol 命令 print 对应于 SML 操作码 11(代表“写数据”)。后一部分为操作码执行操作时所需要的数据在内存中存放的位置。编译器会查找字符表 symboltable 以确定每一个标识符在内存中的位置,这个位置用于完成 SML 指令。

每一条 Simple 语句的编译都基于其命令。例如:一条 rem 语句的行号插入字符表后,剩下的就会被编译器忽略,因为 rem 后的部分仅仅是注释。命令 input, print, goto 和 end 相当于 SML 中的指令 read, write, branch 和 halt。包含这些 Simple 命令的语句能够直接被转换为 SML(注意:如果 goto 语句要转向其后的行,那么编译可能会产生一个不完整的引用,这有时被称为“提前引用”)。

goto 语句编译为不完整的引用时,SML 指令必须注有标记,以表明编译器的第二次编译必须完成这条指令。标记可以存放在一个有 100 个元素的 int 类型的数组 flags 里,每个数组元素都被初始化为 0。如果在 Simple 程序里,行号表示的位置还不确定(也就是说,这个行号在字符表里面不存在),该行号就会存储在数组 flags 里,其

下标与那条不完整的指令的下标相同。不完整的指令的操作数暂时设置为 0。例如,在编译器执行第二次编译之前,无条件分支指令(它将导致一个“提前引用”)会被设为 +4000。稍后会简要描述第二次编译。

if/goto 和 let 语句的编译比其他语句复杂。只有它们能够产生多于一条的 SML 指令。对 if/goto 语句来说,编译器产生测试分支条件的代码,并在必要的情况下将程序转向另一行语句。分支的结果可能是一个不完整的引用。每一个关系操作符能够用 SML 的指令 branch zero 和 branch negative(或者两者的联合)来模拟。

对 let 语句来说,编译器将生成代码计算包括了整型变量和常量的复杂代数表达式。表达式应该用空格来分隔操作符和操作数。练习题 15.12 和练习题 15.13 表述了将中缀表达式转换为后缀表达式的算法以及被编译器用于计算表达式的后缀表达式计算算法。在继续构建编译器之前,应该完成这些练习。编译器遇到表达式时,会将表达式从中缀表达转换为后缀表达,然后计算后缀表达式。

那么,编译器是怎样生成机器代码计算包含了变量的表达式呢? 后缀表达式算法包括了一个“异常指令”(hook),通过“异常指令”,编译器能够产生 SML 指令,而不是真正地计算表达式。为了在编译器里使用“异常指令”,必须修改后缀表达式算法,在字符表里查找它所遇到的每个标识符(还可能将它插入到字符表),然后确定字符在内存中的位置,并将其压入堆栈(代替这个标识符)。当在后缀表达式里遇到一个操作符的时候,处在栈顶的两个内存位置就被弹出,然后将内存位置当作操作数来生成机器代码。表达式计算的每一步结果都被存放在内存中的一个临时位置,同时也要将它们压入堆栈,以便后缀表达式的计算能够继续进行。后缀表达式计算完成的时候,包含了计算结果的内存位置便是惟一存在于堆栈中的元素。然后它被弹出,同时产生 SML 指令将其赋给 let 语句左边的变量。

## 第二次编译

编译器的第二次编译要完成二项任务:解决不完整的引用以及将 SML 代码输出到文件。解决不完整的引用需要以下步骤:

- 查找数组 flags 找出不完整的引用(也就是说数组中值不同于 -1 的元素);
- 在数组 symbolTable 中找到相应的对象,它包含了数组 flags 里面表示不完整引用的标识符(确保标识符的类型为“L”);
- 将找到的 symbolTable 中的 tableEntry 对象的内存位置(即成员 location)插入到带有不完整的引用的指令中(记住包含一个不完整的引用的指令的操作数为“00”);
- 重复以上 3 个步骤,直到数组 flags 结束。

当此过程结束之后,包含 SML 代码的整个数组便输出到磁盘上的一个文件,其每一行为一条 SML 指令。这个文件能够被 Simpletron 在执行的时候读取(必须修改模拟器以便能从文件中读取数据)。现在将你的第一个 Simple 程序编译成 SML 文件,然后在模拟器中执行该文件。你会获得真正的个人成就感。

## 完整示例

下面的例子演示了将一个 Simple 程序转换为 SML 的全过程,该示例程序输入一个整数并且计算从 1 到这个整数的所有整数的和。程序以及编译器第一次编译所产生的 SML 指令如图 15.25,字符表如图 15.26。

| 简单程序                  | SML 位置和指令 | 描述                |
|-----------------------|-----------|-------------------|
| 5 rem sum 1 to x      | 无         | 忽略 rem 行          |
| 10 input x            | 00 +1099  | 将 x 读入到位置 99 中    |
| 15 rem check y == x   | 无         | rem 行被忽略          |
| 20 if y == x goto 60  | 01 +2098  | 02 +3199          |
|                       | 03 +4200  | 将 y ( 98 )加载到累加器中 |
|                       |           | 从累加器中减去 x(99)     |
| 30 let y = y + 1      | 04 +2098  | 将 y 加载到累加器        |
| 05 +3097              |           | 将 1(97)加入累加器      |
|                       | 06 +2196  | 将结果存储在临时位置 96     |
|                       | 07 +2096  | 从临时位置 96 载入数据     |
|                       | 08 +2198  | 将累加器中的值保存在 y 中    |
| 35 rem add y to total | 无         | rem 行被忽略          |
| 40 let t = t + y      | 09 +2095  | 将 t(95)加载进累加器     |
|                       | 10 +3098  | 将 y 加进累加器         |
|                       | 11 +2194  | 将结果储存在临时位置 94     |
|                       | 12 +2094  | 从临时位置 94 载入数据     |
|                       | 13 +2195  | 将累加器中的值保存在 t 中    |
| 45 rem loop y         | 无         | 忽略 rem 行          |
| 50 goto 20            | 14 +4001  | 转到位置 01           |
| 55 rem output result  | 无         | 忽略 rem 行          |
| 60 print t            | 15 +1195  | 将 t 输出到屏幕         |
| 99 end                | 16 +4300  | 程序终止              |

图 15.25 编译器第一次编译产生的 SML 指令



(续表)

| 字符  | 数据类型 | 位置 |
|-----|------|----|
| 15  | L    | 01 |
| 20  | L    | 01 |
| 'Y' | V    | 98 |
| 25  | L    | 04 |
| 30  | L    | 04 |
| 1   | C    | 97 |
| 35  | L    | 09 |
| 40  | L    | 09 |
| 't' | V    | 95 |
| 45  | L    | 14 |
| 50  | L    | 14 |
| 55  | L    | 15 |
| 60  | L    | 15 |
| 99  | L    | 16 |

图 15.26 图 15.25 所示程序中采用的字符表

绝大多数 Simple 语句能够直接转换为单个 SML 指令。在程序中惟一例外的是注释,第 20 行的 if/goto 语句以及程序中所有的 let 语句。注释并不会转换为机器代码。然而,注释的行号会放入字符表中,以免其行号被 goto 语句或 if/goto 语句引用。第 20 行的语句表明如果条件  $y == x$  为真的话,程序控制就会被转到第 60 行。因为第 60 行出现在程序的后面,此时编译器的第一次扫描还没有将行号 60 放在字符表中(只有当语句的行号出现在一条语句的开头时,它才会被放入字符表中),因此,此时不可能决定 SML 中行 3 的指令 branch zero 的操作数。编译器将 60 放在数组 flags 里的位置 03 以表明需要第二次编译来完成这条指令。

因为 Simple 语句与 SML 指令并没有一一对应,我们必须追踪下一条指令在数组 SML 中的位置。例如,第 20 行的 if/goto 语句被编译成 3 条 SML 指令。每产生一条指令,我们就必须将指令计数器加 1 以指向数组 SML 的下一条指令。注意,对拥有较多语句、变量和常量的程序,Simpletron 内存的大小可能会出现问题,因为编译器可能耗尽内存。为了避免这种情况,程序中应该包括一个数据计数器 data counter 用于跟踪下一个变量或常量在数组 SML 中的位置。如果指令计数器 instruction counter 的值比数据计数器 data counter 的值大,数组 SML 将会溢出。此时,编译器应该终止编译过程并打印消息表明内存消耗殆尽。以上所述是为了强调:虽然程序员不需要管理编译器的内存,但编译器本身必须确定内存中指令与数据的位置,并且在编译的过程中当内存被耗尽的时候能够检查到这种错误。

#### 编译过程各步骤详解

接下来看图 15.25 中 Simple 程序的编译过程。首先编译器从程序中将第一行读

入内存,语句

```
5 rem sum 1 to x
```

的第一个标识符(行号)可以用 `strtok` 来确定(参见第 5 章和第 16 章关于 C++ 字符操作函数的讨论),`strtok` 返回的标识符可以用 `atoi` 转换为整型,因此标识符 5 就能够在字符表中定位。如果标识符在字符表中不存在,则将其插入字符表。因为这是程序的第一行,字符表为空,所以 5 就被插入字符表,同时其类型 L 与其在 SML 数组中的位置 00 也存放在一起。尽管这一行为注释,字符表仍然为其分配了行号存储空间来保存行号(以防止被 `if/goto` 和 `goto` 语句引用)。由于 `rem` 语句不会生成指令,因此指令计数器没有递增。

接下来,语句

```
10 input x
```

被分解为标识符。行号 10、其类型 L 及在 SML 数组中的位置(其值为 00,因为程序是以注释行开始的,所以指令计数器的当前值仍为 00)存放于字符表。因为命令 `input` 直接对应于一条 SML 操作码,所以编译器必须确定它在 SML 数组中的位置。但字符表中找不到 `x`,所以 `x` 便插入字符表,包括字母 `x` 的 ASCII 值,类型 V 以及它在 SML 数组中的位置 99(数据储存从位置 99 开始,向后存放)。现在就能产生这条语句的 SML 代码:将操作码 10(SML 的读操作码)乘以 100,再增加 `x` 的位置(在字符表中可以确定)即可。然后指令计数器增 1 以表示生成了一条指令。

接着,语句

```
15 rem check y == x
```

被分解为标识符。编译器查找字符表来寻找行号 15(但没有找到)。所以行号 15 同它的类型 L 及位置 01 被插入字符表(记住 `rem` 语句并没有生成指令,所以计数器没有递增)。

随后,语句

```
20 if y == x goto 60
```

被分解为标识符。行号 20、其类型 L 及位置 01 就被插入到字符表中。命令 `if` 表明有条件被判断。在字符表中找不到变量 `y`,因此 `y` 就与其类型 V 和位置 98 插入字符表。接下来就要生成指令以判断条件。因为在 SML 中没有等价的关系操作符,必须对 `x` 和 `y` 进行计算然后再根据结果进行分支以实现模拟。如果 `y` 等于 `x`,即 `y` 减去 `x` 的结果为 0,因此就可使用 `branch zero` 来模拟 `if/goto` 语句。这就生成了指令 +3199。累加器中的值可能为 0,负或正。因为操作符为 `==`,就可使用 `branch zero`。首先,查找字符表来寻找分支转向的位置(这里为 60),但是找不到。因此 60 就保存在数组 `flags` 的位置 03,同时生成指令 03 + 4200(此时还不能增加分支定位,因为我们还没有在数组 SML 中为第 60 行分配位置)。现在指令计数器递增到 04。

编译器接着处理语句

```
25 rem increment y
```

此时,行号 25、其类型 L 和位置 04 被插入字符表。指令计数器没有递增。

当语句

```
30 let y = y + 1
```

被分解为标识符时,行号 30、其类型 L 和位置 04 被插入字符表。命令 let 表明这是一条赋值语句。所有标识符都被插入字符表(如果它们还不存在的话)。整数 1 与其类型 C、位置 97 被插入字符表。接下来,编译器要将等号右边的表达式从中缀表达转换为后缀表达,然后计算后缀表达式( $y\ 1\ +$ )。编译器在字符表定位变量 y,并将其在内存中的位置压入堆栈。遇到操作符 + 时,后缀表达式计算子程序先从堆栈中弹出操作符的右操作数,然后再弹出左操作数,于是便产生了指令

```
04 +2098 (load y)
05 +3097 (add 1)
```

表达式的计算结果存放在内存中的临时位置(96),指令为

```
06 +2196 (store temporary)
```

同时临时位置被压入堆栈。因为表达式已经完成计算,其结果就可以保存到变量 y (也就是 = 左边的变量)。所以临时位置的值被载入累加器,并存放在变量 y 中,指令为

```
07 +2096 (load temporary)
08 +2198 (store y)
```

你马上便能发现这些指令是多余的。有关问题,参见稍后的讨论。

当语句

```
35 rem add y to total
```

被分解为标识符时,行号 35 与其类型 L、位置 09 就被插入字符表。

语句

```
40 let t = t + y
```

与行 30 类似。变量 t 与其类型 V、位置 95 被插入字符表。产生指令的方法也与第 30 行相同,指令为 09 + 2095, 10 + 3098, 11 + 2194, 12 + 2094 和 13 + 2195。注意,  $t + y$  的结果在被赋给变量 t(95)之前存放于内存中的临时位置。这里,你会发现存在多余的指令。

语句

```
45 rem loop y
```

是一行注释,因此行号 45 与其类型 L、位置 14 被插入字符表。

语句

```
50 goto 20
```

将程序控制转向第 20 行。行号 50 与其类型 L、位置 14 被插入字符表。goto 命令在 SML 中相当于指令 unconditional branch(40),它将程序控制转到 SML 指定的位置。现在编译器在字符表中查找行号 20,找到其对应位置为 01。然后编译器将操作码 40 乘以 100,并加入位置 01 从而生成指令 14 + 4001。

语句

```
55 rem output result
```

是一行注释,因此行号 55 与其类型 L、位置 15 被插入字符表。

语句

```
60 print t
```

是一条输出语句。行号 60 与其类型 L、位置 15 被插入字符表。命令 print 对应于

SML 里的操作码 11(即写指令 read)。字符表中可以找到变量 *t* 的位置,然后同操作码乘以 100 的结果一起生成指令。

语句

99 end

是程序的最后一行。行号 90 与其类型 L、位置 16 被插入到字符表中。命令 end 所产生的 SML 指令为 +4300(操作码 43 对应于 SML 中的指令 halt),这条指令是 SML 内存数组中的最后一条指令。

编译器的第一次编译到此结束。现在来看第二次编译。编译器首先查找数组 flags 来寻找值不同于 -1 的元素。数组 flags 的位置 03 上的元素值为 60,因此编译器知道指令 03 是不完整的。然后编译器查找字符表以确定 60 的位置,并将查找结果加到不完整的指令中,使之完整。在上述的程序中,编译器在字符表中可以找到行号 60 的位置 15,因此生成指令 03 + 4215 代替不完整的指令 03 + 4200。现在,这个 Simple 程序便得以成功编译。

为了构建这个编译器,必须完成如下任务:

- a) 修改练习题 5.19 中的程序以便 Simpletron 模拟器能够 from 被用户指定的文件中读入数据(参见第 14 章)。模拟器要以与输出到屏幕上相同的格式将结果输出到磁盘上的文件。同时要将修改这个模拟器使之面向对象。特别地,使硬件的每一部分都成其为一个对象。可以使用继承将指令类型构成一个类的层次结构,然后发出消息 execute Instruction 来让每一条指令执行其本身从面多态地执行程序。
  - b) 修改练习题 15.12 中的中缀表达式到后缀表达式的转换算法,使之能够处理多位的整型操作数和单字母的变量操作数(提示:可以使用标准库函数 strtok 来确定表达式中的每一个常量与变量,同时也可使用标准库函数 atoi 将字符串转换为整型值)。注意:必须改变后缀表达式的数据表达法使之能够支持变量名与整型常量。
  - c) 修改后缀表达式计算算法使之能够处理多位的整型操作数和变量操作数。同时也要实现前面所述的“异常处理指令”来产生相应的 SML 指令而不是直接计算后缀表达式(提示:可以使用标准库函数 strtok 来确定表达式中的每一个常量与变量,同时也可使用标准库函数 atoi 将字符串转换为整数)。注意:必须改变后缀表达式的数据表达法使之能够支持变量名与整型常量。
  - d) 构建你的编译器。使用前 2 个步骤来计算程序中 let 语句所包含的表达式。程序应该包含一个执行第一次编译的函数和一个执行第二次编译的函数。二个函数都能调用其他的函数来执行它们的任务。注意尽量使用面向对象的方法来实现。
- 15.28 (优化 Simple 编译器)程序经过编译并转换为 SML 后,便生成了一系列指令。部分特定组合的指令往往会多次出现,并通常以 3 个一组的形式出现,这种指令称为结果生成指令(production)。结果生成指令通常包括 3 个指令如 load, add 和 store。例如,图 15.27 为图 15.25 中程序在编译过程中所产生的 5 条指令。前 3 条指令是将 *y* 与 1 相加的一个结果生成指令。注意,指令 06 和 07 将累加器的值放入内存的临时位

置,然后又将其载入累加器,最后指令 08 将其保存到变量中。结果生成指令后往往会跟一个装载指令 load,将刚刚保存的数据重新置于内存。因此能够通过删除结果生成指令中的 store 指令及其后的 load 指令来优化代码,使 Simpletron 能够更快地执行程序。图 15.28 为 15.25 中代码优化后的结果。我们可以看到,优化后的代码共减少了 4 条指令,也即节省了 25% 的内存空间。

现在修改编译器以提供优化生成的 Simpletron 机器代码的选项。然后手工比较优化前后的代码,计算代码减少的百分比。

15.29 (修改 Simple 编译器)对 Simple 编译器进行下列修改。有些修改可能涉及到修改练习题 5.19 中的 Simpletron 模拟器。

- a) 允许在 let 语句中使用求模操作符 (%)。此时还需要修改 Simpletron 模拟器以包含求模指令。
- b) 允许在 let 语句中使用幂操作符 (^)。此时还需要修改 Simpletron 模拟器以包含幂运算指令。
- c) 使编译器可以识别 Simple 语句中的大小写(如“A”与“a”等价)。此时不需要修改模拟器。
- d) 使 input 语句可以获取多个变量的值,如 input x, y。此时不需要修改模拟器。
- e) 使编译器可以在单条 print 语句中输出多个变量的值,如 print a, b, c。此时不需要修改模拟器。
- f) 增加语法错误检查。Simple 程序中遇到语法错误时,能显示错误消息。此时不需要修改模拟器。
- g) 允许使用整型数组。此时不需要修改模拟器。
- h) 允许用 Simple 命令 gosub 和 return 调用子程序。命令 gosub 将程序控制权交给子程序,然后 return 将控制权交回调用者内 gosub 命令之后的语句。这与 C++ 的函数类似。同一个子程序能够被多个 gosub 命令调用。此时不需要修改模拟器。
- i) 允许下列形式的循环结构

```
for x=2 to 10 step 2
Simple statements
next
```

for 语句以 2 为步长在 2~10 之间循环。next 行标志着 for 循环体结束。此时不需要修改模拟器。

- j) 允许下列形式的循环结构

```
for x=2 to 10
Simple statements
next
```

for 语句以 1 为步长在 2~10 之间循环。此时不需要修改模拟器。

- k) 允许编译器处理字符串输入与输出。这需要修改 Simpletron 模拟器以便能够处理和保存字符串。(提示:每个 Simpletron 字分为两部分,每部分都包含一个 2 位的整数,代表着一个字符的 ASCII 值的 10 进制值。增加一条机器指令

以打印从 Simpletron 内存指定位置开始的字符串。在所指定位置的字中,前半部分代表字符串中字符的数目,即字符串的长度。之后的每半个字都代表着一个字符。机器指令校验长度后再分别将 2 位的整数转换为字符然后打印它们。)

- 1) 允许编译器除了处理整型数据外,还能处理浮点数据。Simpletron 模拟器必须修改以使用于处理浮点数据。

|   |    |           |
|---|----|-----------|
| 1 | 04 | +2098(载入) |
| 2 | 05 | +3097(加)  |
| 3 | 06 | +2196(保存) |
| 4 | 07 | +2096(载入) |
| 5 | 08 | +2198(保存) |

图 15.27 图 15.25 所示程序中未经优化的代码

| Simple 程序             | SML 内存位置及指令 | 描述                             |
|-----------------------|-------------|--------------------------------|
| 5 rem sum 1 to x      | 无           | 忽略 rem 行                       |
| 10 input x            | 00 +1099    | 将 x 读入位置 99                    |
| 15 rem check y == x   | 无           | 忽略 rem 行                       |
| 20 if y == x goto 60  | 01          | +2098 将 y ( 98 ) 载入累加器         |
|                       | 02          | +3199 从累加器中减去 x(99)            |
|                       | 03          | +4200 用命令 branch zero 转到未确定的位置 |
| 30 let y = y + 1      | 04 +2098    | 将 y 载入累加器                      |
|                       | 05 +3097    | 将 1(97)加进累加器                   |
|                       | 06 +2198    | 将累加器中的值保存在 y 中                 |
| 35 rem add y to total | 无           | 忽略 rem 行                       |
| 40 let t = t + y      | 07 +2095    | 将 t(95)载入累加器                   |
|                       | 08 +3098    | 将 y 加入累加器                      |
|                       | 09 +2195    | 将累加器中的值保存在 t 中                 |
| 45 rem loop y         | 无           | 忽略 rem 行                       |
| 50 goto 20            | 10 +4001    | 转到位置 01                        |
| 55 rem output result  | 无           | 被忽略 rem 行                      |
| 60 print t            | 11 +1195    | 将 t(96)输出到屏幕                   |
| 99 end                | 12 +4300    | 程序终止                           |

图 15.28 图 15.25 所示程序中已经优化的代码

- 15.30 (一个 Simple 解释器)解释器是一种程序读取高级语言语句,然后确定并立即执行语句要执行的操作。高级语言起初并没有被转换为机器语言。解释器执行程序很慢,因为在程序中的每一条语句都要先执行解码。如果语句包含在循环中,循环每进行一次,就要对它们进行一次解码。BASIC 语言的早期版本就是解释执行的。为练习题 15.26 中的 Simple 语言编写一个解释器。程序应该使用练习题 15.12 中编

写的程序(将中缀表达式转换到后缀表达式)以及练习 15.13 题中编写的后缀表达式计算程序来计算 let 语句中的表达式。同时也要遵守练习题 15.26 提出的限制。比较这些程序在解释器内的运行结果和编译 Simpletkyer 结果,并在练习题 5.19 创建 Simpletron 模拟器中运行它们。

- 15.31 (在链表任意位置插入/删除)我们的链表类模板只允许在链表的头部和尾部进行插入和删除操作。重用链表类模板使用 private 继承与构造仅用最少的代码来创建堆栈类模板与队列类模板时,这样的确方便。但实际上,链表比我们所描述的更简单。修改本章提出的链表类模板使之能够在链表任意位置进行插入与删除操作。
- 15.32 (无追尾指针的链表与队列)在图 15.3 中的链表类中,我们使用了两个指针,即 firstPtr 和 lastPtr。在我们执行链表类 List 的操作 insertAtBack 与 removeFromBack 时,追尾指针 lastPtr 是有用的。函数 insertAtBack 对应于队列类 Queue 的成员函数 enqueue。重写链表类 List 使之不使用追尾指针 lastPtr,因此任何对链表尾部的操作都只能从链表头部开始向后查找。由此引发的问题是:这是否会影响队列类 Queue 的实现?
- 15.33 用图 15.11 堆栈程序的合成版本来编写一个可完全运行的堆栈程序。然后把该程序修改为内联(inline)成员函数。比较这两种方法,归纳内联成员函数的优缺点。
- 15.34 (二叉树排序与查找的性能)二叉树排序存在着这样的问题:即使使用相同的数据,只要数据插入二叉树的次序不同,二叉树的形状也会不同。而二叉树排序与查找算法对树的形状很很大的关系。如果数据以递增的次序插入树,二叉树的形状会如何?以递减的次序又会如何?在何种形状下,二叉树的查找性能最佳?
- 15.35 (索引链表)在正文中我们曾经谈到,链表必须按顺序来查找。对大型链表来说,这样做,性能可能较低。改善查找性能的常见作法之一是创建并维护链表的索引。索引是链表中一些不同的关键位置的指针的集合。例如,一项查找一大串入名的应用就可以通过增加其中包含有 26 项(每一项代表着字母表中的一个字母)的索引来改进查找性能。这样,查找以“Y”开头的姓的人名时就可以先确定索引中“Y”项开始的位置,然后从此位置起进行线性查找直到找到需要的人名为止。这样做比直接从链表头开始查找快得多。使用图 15.3 中的链表类 List 来作为索引链表类 IndexedList 的基类。编写程序演示索引链表的操作。程序应该包括成员函数 insertInIndexedlist, searchIndexedlist 和 deleteFromIndexedlist。

# 第 16 章 位、字符、字符串与结构

## 学习目标

- 能够创建和使用结构
- 能够通过按值传递和按引用传递将结构参数传递给函数
- 能够使用位操作符来操纵数据以及创建位段来紧密地存储数据
- 能够使用字符处理库(cctype)中的函数
- 能够使用通用函数库(cstdlib)中的字符转换函数
- 能够使用字符串处理函数库(cstring)中的函数
- 理解利用函数库可实现软件重用

## 16.1 简介

本章首先要介绍结构,然后要讨论位、字符、字符串的操作。其中包含的许多技巧与 C 类似。将它们归为本章是考虑到那些要使用 C 遗留代码的 C++ 程序员的需要。

结构可以包含不同数据类型的多个变量,而数组中所有元素的类型都是相同的。这个特性以及我们在下而要讨论的结构相关内容同样适用于类。在 C++ 中,类与结构的主要差别是类成员在默认状态下是私有访问的,结构成员在默认状态下是公有访问的。结构通常用于定义存储在文件中的数据结构(参见第 14 章)。指针与结构有利于形成更复杂的数据结构如链表、队列、堆栈与树(参见第 15 章)。下面我们将讨论如何声明和初始化结构以及如何将结构传递给函数,随后会介绍一个高效的洗牌与发牌模拟程序。

## 16.2 结构的定义

### 结构定义

```
struct Card {  
    char *face;  
    char *suit;  
};
```

中,关键字 struct 用于定义结构 Card。标识符 Card 是结构名,在 C++ 中用来声明结构类型变量(在 C 中,上面定义的结构为 struct Card 类型)。本例中,结构类型为 Card。在结构定义中花括号中的数据(可能为函数,就像类一样)为结构成员。同一结构中的成员名必须是惟一的,不同结构中成员名可以相同但不会引起冲突。结构定义必须以分号结束。

**常见编程错误 16.1** 结构定义结束处忘记加分号。



结构 Card 包括两个类型为 `char *` 的成员: `face` 与 `suit`。结构成员可能是基本数据类型的变量(如 `int` 和 `double` 等等)或者是集合如数组和其他结构。第 4 章中我们看到,数组中每个元素的类型都必须相同。然而结构的数据成员却可能是多种数据类型。例如,一个 `Employee` 结构中,有代表雇员姓名的字符串成员,代表雇员年龄的 `int` 成员,代表性别的 `char` 成员,以及代表雇员工资的 `double` 成员等。

结构不能包含其本身。例如,一个类型为 `Card` 的结构变量不能包含于结构 `Card` 的定义中,但指向 `Card` 结构的指针能够包含 `Card` 结构的定义中。如果一个结构内包含了一个指向与其类型相同的另一个结构的指针成员,那么它就是“自引用结构”。自引用结构在第 15 章用于构建不同类型的链接数据结构。

前而定义的结构并没有占用内存空间,它只是创建了一种用于声明结构变量的新的数据类型。结构变量声明与其他类型的变量声明相同。定义

```
Card oneCard, deck[52], *cPtr;
```

把 `oneCard` 声明为 `Card` 类型的结构变量,把 `deck` 声明为 `Card` 类型的、拥有 52 个元素的数组,声明 `cPtr` 为指向 `Card` 结构的指针。指定结构的变量声明可以通过将变量名放在结构定义的右花括号与分号之间来进行。例如,前面声明可以合并为

```
struct Card {
    char *face;
    char *suit;
} oneCard, deck[52], *cPtr;
```

结构名是可选的。如果不包含结构名,结构类型的变量就只能在结构定义中进行声明,不能进行其他单独声明。

**良好编程习惯 16.1** 创建结构时始终提供结构名。这样一来,在程序后面定义这种结构类型的变量时就会很方便。将结构作为参数传递给函数时,结构名是必须的。

结构中惟一可用的内部操作包括:将结构赋给同样类型的结构,取结构的地址,访问结构成员(参见第 6 章)以及用操作符 `sizeof` 来确定结构的大小。与类一样,在结构对象中,很多操作符也可以重载。

结构成员在内存中并不是按字节连续存放的。有时在结构中会留有“空洞”,因为计算机只能沿某种内存边界(比如半字、一个字或双字边界)来保存指定类型的数据。一个字是计算机中存储数据的标准单元,通常为 2 个字节或 4 个字节。结构定义

```
struct Example {
    char c;
    int i;
} sample1, sample2;
```

中也定义了结构的两个变量 `sample1`, `sample2`, 一个字为两个字节的计算机可能要求每个 `Example` 成员按字的边界对齐。也就是说,在字的开头对齐(这与机器有关)。图 16.1 为一个类型为 `Example` 的对象(其中 `c` 赋值为“a”,`i` 赋值为 97)在内存中的存储情况(数据按二进制值来显示)。我们可以看到,在类型为 `Example` 的对象的存储空间中留了一个字节的“空洞”,这个“空洞”的数据是没有经过定义的。如果 `sample1` 与 `sample2` 的成员值相同,它

们的比较结果也不一定相同,因为没有定义的一个字节的“空洞”不可能包含相同的数据。

**常见编程错误 16.2** 比较结构会导致语法错误,因为不同系统的边界对齐要求不同。

**可移植性提示 16.1** 因为特定数据类型的大小与机器有关,以及存储边界对齐也与机器有关,所以结构的数据表示法也与机器有关。

## 16.3 结构的初始化

如同数组一样,结构能用初始化数据列表来进行初始化。要想初始化结构,可在其结构变量声明后放置一个等号,然后放置用花括号括起来的初始化数据。例如,定义

```
Card oneCard = { "Three", "Hearts" };
```

创建了结构 Card(在前面已经定义)的变量 oneCard 并且将其成员 face 和 suit 分别初始化为“Three”和“Hearts”。如果初始化数据列表少于成员数,剩下的成员就会初始化为 0。在函数定义之外声明的结构变量,如果在其外部声明中没有显式初始化,其成员也将初始化为 0。结构变量还可以通过下列方式来初始化:将另一个同样类型的结构变量赋给它;将值分别赋给结构的单个数据成员。

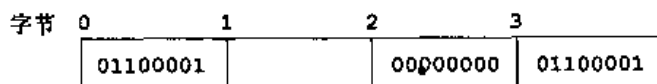


图 16.1 为 Example 类型变量分配的存储空间,展示了尚未定义的内存区域

## 16.4 在函数中使用结构

可通过两种方式将结构中的信息传递给函数的。你可以把整个结构传递给函数或只把结构的单个成员传递给函数。默认情况下,它们是按值传递的(单个数组元素除外)。结构及其成员可用引用或指针进行按引用传递。

要想按引用传递结构,可将结构变量的地址或引用传递给结构变量。与其他数组一样,结构数组是按引用传递的。

第 4 章曾提到,数组能够用结构进行按值传递。要想这样,先创建一个结构(或类),并将数组作为其成员。因为结构是按值传递的,所以数组也会按值传递。

**常见编程错误 16.3** 认为结构像数组那样是按引用传递的,并试图在被调用函数中修改调用函数中结构的值。

**性能提示 16.1** 按引用传递结构比按值传递结构(这需要复制整个结构)更有效率,尤其是大型结构。

## 16.5 关键字 typedef

关键字 typedef 能够为在前面定义的数据类型创建同义名(或别名)。人们常用 typedef

来定义更短小、可读性更强的类型名。例如,语句

```
typedef Card * CardPtr
```

为类型 `Card *` 定义了一种新的类型名 `CardPtr`。

**良好编程习惯 16.2** 对 `typedef` 定义的新类型名采用大写以表明它们是其他类型名的同义名或别名。

用 `typedef` 定义一种新的类型名并没有创建新的数据类型。`typedef` 只是创建了一个新的类型名,它可用作现有类型的别名。

内置数据类型的别名可以用 `typedef` 来创建。例如,一个需要 4 个字节整数的程序可能会在一个系统中使用类型名 `int`,在另外一个整数为 2 个字节的系统内使用类型名 `long int`。为了提高程序的可移植性,程序可用 `typedef` 关键字为 4 个字节的整数创建别名如 `Integer`。因此,`Integer` 既可以用作整数为 4 个字节的系统中 `int` 类型的别名,也可用作整数为 2 个字节的系统中 `long int` 类型的别名。这样一来,为了提高可移植性,程序员只须在程序里用 `Integer` 来声明所有 4 个字节的整数变量。

**可移植性提示 16.2** 使用 `typedef` 可增加程序的移植性。

## 16.6 示例:高性能洗牌与发牌模拟程序

图 16.2 中的程序建立在第 5 章洗牌与发牌模拟程序基础上。程序将一副扑克牌定义为一个结构数组并且使用了高性能洗牌与发牌算法,程序输出如图 16.3 所示。

```
1 //Fig.16.2: fig16_02.cpp
2 //Card shuffling and dealing program using structures
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::ios;
9
10 #include <iomanip>
11
12 using std::setiosflags;
13 using std::setw;
14
15 #include <cstdlib>
16 #include <ctime>
17
18 struct Card {
19     char *face;
20     char *suit;
21 };
22
23 void fillDeck( Card * const, char *[], char *[] );
24 void shuffle( Card * const );
```

```

25 void deal( Card * const );
26
27 int main()
28 {
29     Card deck[ 52 ];
30     char *face[] = { "Ace", "Deuce", "Three", "Four",
31                     "Five", "Six", "Seven", "Eight",
32                     "Nine", "Ten", "Jack", "Queen",
33                     "King" };
34     char *suit[] = { "Hearts", "Diamonds",
35                     "Clubs", "Spades" };
36
37     srand( time( 0 ) );           //randomize
38     fillDeck( deck, face, suit );
39     shuffle( deck );
40     deal( deck );
41     return 0;
42 }
43
44 void fillDeck( Card * const wDeck, char *wFace[],
45              char *wSuit[] )
46 {
47     for ( int i = 0; i < 52; i++ ) {
48         wDeck[ i ].face = wFace[ i % 13 ];
49         wDeck[ i ].suit = wSuit[ i / 13 ];
50     }
51 }
52
53 void shuffle( Card * const wDeck )
54 {
55     for ( int i = 0; i < 52; i++ ) {
56         int j = rand() % 52;
57         Card temp = wDeck[ i ];
58         wDeck[ i ] = wDeck[ j ];
59         wDeck[ j ] = temp;
60     }
61 }
62
63 void deal( Card * const wDeck )
64 {
65     for ( int i = 0; i < 52; i++ )
66         cout << setiosflags( ios::right )
67              << setw( 5 ) << wDeck[ i ].face << " of "
68              << setiosflags( ios::left )
69              << setw( 8 ) << wDeck[ i ].suit
70              << ( ( i + 1 ) % 2 ? '\t' : '\n' );
71 }

```

图 16.2 高性能的洗牌与发牌模拟程序

输出结果:

|                   |                   |
|-------------------|-------------------|
| Eight of Diamonds | Ace of Hearts     |
| Eight of Clubs    | Five of Spades    |
| Seven of Hearts   | Deuce of Diamonds |
| Ace of Clubs      | Ten of Diamonds   |
| Deuce of Spades   | Six of Diamonds   |
| Seven of Spades   | Deuce of Clubs    |
| Jack of Clubs     | Ten of Spades     |
| King of Hearts    | Jack of Diamonds  |
| Three of Hearts   | Three of Diamonds |
| Three of Clubs    | Nine of Clubs     |
| Ten of Hearts     | Deuce of Hearts   |
| Ten of Clubs      | Seven of Diamonds |
| Six of Clubs      | Queen of Spades   |
| Six of Hearts     | Three of Spades   |
| Nine of Diamonds  | Ace of Diamonds   |
| Jack of Spades    | Five of Clubs     |
| King of Diamonds  | Seven of Clubs    |
| Nine of Spades    | Four of Hearts    |
| Six of Spades     | Eight of Spades   |
| Queen of Diamonds | Five of Diamonds  |
| Ace of Spades     | Nine of Hearts    |
| King of Clubs     | Five of Hearts    |
| King of Spades    | Four of Diamonds  |
| Queen of Hearts   | Eight of Hearts   |
| Four of Spades    | Jack of Hearts    |
| Four of Clubs     | Queen of Clubs    |

图 16.3 高性能洗牌与发牌模拟程序的输出结果

在程序中,函数 fillDeck 按每种花色从 A~K 的顺序,对 Card 数组进行了初始化。然后 Card 数组被传递给函数 shuffle(高性能洗牌算法通过该函数来执行)。函数 shuffle 将一个拥有 52 个 Card 结构的数组作为参数。该函数在所有的 52 张牌中进行循环(数组下标从 0~51)。对每一张牌来说,函数首先在 0~51 之间随机挑选一个数,然后将当前的 Card 结构与随机挑选的 Card 结构进行交换。在整个数组的一次循环中共要进行 52 次交换。这样,Card 结构数组中的元素就被随机地混和在一起,也即洗牌完毕。这个算法并没有第 5 章中表述的算法的那种不确定的延迟。因为 Card 结构数组是按位置交换的,在函数 deal 中高效发牌算法只需要循环一次,就可发完所有洗好了的牌。

常见编程错误 16.4 引用结构数组中的单个结构时,忘记包含数组下标。

## 16.7 位操作符

针对那些需要执行所谓“位和字节”操作的程序员,C++ 提供了广泛的位操作能力。操作系统、设备测试软件、网络软件以及其他许多软件都需要程序员直接操作硬件。在本节及随后几节里,我们将讨论 C++ 的位操作能力。在简要介绍 C++ 的众多位操作符的同时,还要讨论如何用位段来节约内存。

在计算机内部,所有数据都表示为位的序列。每一位要么取 0,要么取 1。在大多数系统中,8 位形成一个字节,也即一个类型为 `char` 的变量的标准存储空间。其他的数据类型则需要更多的字节来存放。位操作符通常用来操作整型操作数(类型为 `char`, `short`, `int` 和 `long`, 以及 `signed` 和 `unsigned`)的位数据。位操作符通常与无符号整数一起使用。

**可移植性提示 16.3** 位数据操作与机器硬件有关。

注意,本章的位操作符讨论均用整型操作数的二进制数据表示(有关二进制详情参见附录 C“数值系统”)。因为位操作与机器有关,本章部分程序可能无法在你的机器上运行。

位操作符有:位与 `&`、位或 `|`、位异或 `^`、位左移 `<<`、位右移 `>>` 和取反 `~` (注意,我们一直将 `&`, `<<`, `>>` 用作别的用途,这是操作符重载的典型例子)。位与 `&`、位或 `|`、位异或 `^` 按位来比较两个整型操作数。位与 `&` 只有在两个操作数对应位全 1 的情况下才将结果位设置为 1。位或 `|` 只要两个操作数对应的位中有 1 个为 1 就将结果位设为 1。位异或 `^` 只有在两个操作数的对应位上仅有 1 个为 1 时才将结果位置为 1。左移操作符 `<<` 将其左边指定的操作数左移其右边的位数。右移操作数 `>>` 将其左边的操作数右移其右边的位数。取反操作符对操作数按位进行取反,即如果操作数位为 1,则将结果位设为 0;反之,如果操作数位为 0,则将结果位设为 1。对每一种位操作符的详细讨论在稍后的例子中演示。位操作符归纳表如图 16.4。

| 操作符                   | 名称  | 说明                               |
|-----------------------|-----|----------------------------------|
| <code>&amp;</code>    | 位与  | 两操作数相应位皆为 1 时,结果位才为 1            |
| <code> </code>        | 位或  | 两操作数中相应位只要有一个为 1,结果位就为 1         |
| <code>^</code>        | 位异或 | 两操作数中相应位仅有一个为 1 时,结果位才为 1        |
| <code>&lt;&lt;</code> | 左移  | 将第一个操作数左移第二个操作数指定的位数;右边以 0 填充    |
| <code>&gt;&gt;</code> | 右移  | 将第一个操作数右移第二个操作数指定的位数;左边填充的数依赖于机器 |
| <code>~</code>        | 取反  | 将 0 置为 1,将 1 置为 0                |

图 16.4 位操作符

使用位操作符进行操作时,打印数据的二进制值以演示这些操作的正确效果是很有用的。图 16.5 以每 8 位为一组按二进制打印了一个 `unsigned` 整数。函数 `displayBits` 使用位与操作符来对变量 `value` 与常量 `MASK` 进行与操作。通常,位与操作符与一个称为“掩码”的整数一起使用。“掩码”是一个特定的整,其位被设为 1 的整数,它用于选择特定的位而屏蔽掉其他位。在函数 `displayBits` 中,“掩码”被赋值为

```
1 << shift
```

常量 `SHIFT` 取值为

```
8 * sizeof(unsigned) - 1
```

```
1 //Fig. 16.5: fig16_05.cpp
2 //Printing an unsigned integer in bits
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
```

```

7
8 #include <iomanip>
9
10 using std::setw;
11 using std::endl;
12
13 void displayBits( unsigned );
14
15 int main()
16 {
17     unsigned x;
18
19     cout << "Enter an unsigned integer: ";
20     cin >> x;
21     displayBits( x );
22     return 0;
23 }
24
25 void displayBits( unsigned value )
26 {
27     const int SHIFT = 8 * sizeof( unsigned ) - 1;
28     const unsigned MASK = 1 << SHIFT;
29
30     cout << setw( 7 ) << value << " = ";
31
32     for ( unsigned c = 1; c <= SHIFT + 1; C++ ) {
33         cout << ( value & MASK ? '1' : '0' );
34         value <<= 1;
35
36         if ( c % 8 == 0 )
37             cout << ' ';
38     }
39
40     cout << endl;
41 }

```

输出结果:

```

Enter an unsigned integer:65000
65000 = 00000000 00000000 11111101 11101000

```

图 16.5 打印按位无符号整数

这个表达式将一个 unsigned 类型的长度乘以 8 得到总的位数后再减去 1。1 左移 SHIFT (即 31) 位的结果为

```
10000000 00000000 00000000 00000000
```

左移操作符中在 MASK 中将 1 从最低位(最右边)移到最高位(最左边),并在右边依次填入 0。语句

```
cout << ( value & MASK ? '1' : '0' );
```

确定打印变量 value 最左边一位的值是 0 或 1。假定变量 value 的值为 65 000(即 00 000 000 00 000 000 11 111 101 11 101 000)。当变量 value 与 MASK 进行与操作时,在变量 value 中最

高位以外的位全部被屏蔽起来,因为任何一位与 0 相与的结果也为 0。如果最左边的位为 1,则 `value & MASK` 结果:

```
00000000 00000000 11111101 11101000    (value)
10000000 00000000 00000000 00000000    (MASK)
-----
00000000 00000000 00000000 00000000    (value & MASK)
```

会被解释为假(false),打印出因此 0。然后用表达式 `value << 1` 将变量 `value` 左移一位(这与 `value = value << 1` 等价)。接着,在变量 `value` 中这些步骤按位重复。最终,有一位值为 1 的位左移到最左边的位置上,位操作如下

```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
-----
10000000 00000000 00000000 00000000    (value & MASK)
```

因为左边位皆为 1,因此打印 1。图 16.6 总结了用位与操作符 `&` 操作两个整数的结果。

| 位 1 | 位 2 | 位 1 & 位 2 |
|-----|-----|-----------|
| 0   | 0   | 0         |
| 1   | 0   | 0         |
| 0   | 1   | 0         |
| 1   | 1   | 1         |

图 16.6 利用位与操作符(`&`)操作两个位的结果

**常见编程错误 16.5** 将位与操作符 `&` 错用作逻辑与操作符 `&&`,或者反之。

图 16.7 中的程序演示了位与、位或、位异或和位取反操作的效果。函数 `displayBits` 打印无符号整型值。程序输出结果如图 16.8 所示。

```
1 //Fig. 16.7: fig16_07.cpp
2 //Using the bitwise AND, bitwise inclusive OR, bitwise
3 //exclusive OR, and bitwise complement operators.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8
9 #include <iomanip>
10
11 using std::endl;
12 using std::setw;
13
14 void displayBits( unsigned );
15
16 int main()
```



```
17 {
18     unsigned number1, number2, mask, setBits;
19
20     number1 = 2179876355;
21     mask = 1;
22     cout << "The result of combining the following\n";
23     displayBits( number1 );
24     displayBits( mask );
25     cout << "using the bitwise AND operator & is\n";
26     displayBits( number1 & mask );
27
28     number1 = 15;
29     setBits = 241;
30     cout << "\nThe result of combining the following\n";
31     displayBits( number1 );
32     displayBits( setBits );
33     cout << "using the bitwise inclusive OR operator | is\n";
34     displayBits( number1 | setBits );
35
36     number1 = 139;
37     number2 = 199;
38     cout << "\nThe result of combining the following\n";
39     displayBits( number1 );
40     displayBits( number2 );
41     cout << "using the bitwise exclusive OR operator ^ is\n";
42     displayBits( number1 ^ number2 );
43
44     number1 = 21845;
45     cout << "\nThe one's complement of\n";
46     displayBits( number1 );
47     cout << "is" << endl;
48     displayBits( ~number1 );
49
50     return 0;
51 }
52
53 void displayBits( unsigned value )
54 {
55     const int SHIFT = 8 * sizeof( unsigned ) - 1;
56     const unsigned MASK = 1 << SHIFT;
57
58     cout << setw( 10 ) << value << " = ";
59
60     for ( unsigned c = 1; c <= SHIFT + 1; c++ ) {
61         cout << ( value & MASK ? '1' : '0' );
62         value <<= 1;
63
64         if ( c % 8 == 0 )
65             cout << ' ';
66     }
67 }
```

```

68     cout << endl;
69 }

```

图 16.7 使用位与、位或、位异或和位取反操作

输出结果:

```

The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
      1 = 00000000 00000000 00000000 00000000
using the bitwise AND operator & is
      1 = 00000000 00000000 00000000 00000001

The result of combining the following
      15 = 00000000 00000000 00000000 00001111
      241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
      255 = 00000000 00000000 00000000 11111111

The result of combining the following
      139 = 00000000 00000000 00000000 10001011
      199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
      76 = 00000000 00000000 00000000 01001100

The one 's complement of
      21845 = 00000000 00000000 01010101 01010101
is
5294945450 = 11111111 11111111 10101010 10101010

```

图 16.8 图 16.7 所示程序的输出结果

在图 16.7 中, 变量 `mask` 赋值为 1 (00000000 00000000 00000000 00000001), 变量 `number1` 赋值为 2 179 876 355 (10000001 11101110 01000110 00000011)。当 `mask` 与 `number1` 在表达式 `number1 & mask` 中用位与 `&` 进行操作时, 结果位为 00000000 00000000 00000000 00000001。与常量 `MASK` 进行与操作使变量 `number1` 中最低位之外的位全被“屏蔽”起来。

位或用于把特定的位设置为 1。在图 16.7 中, 变量 `number1` 赋值为 15 (00000000 00000000 00000000 00001111), 变量 `setBits` 赋值为 241 (00000000 00000000 00000000 11110001)。当这两个变量在表达式 `number1 | setBits` 进行位或操作时, 可得结果 255 (00000000 00000000 00000000 11111111)。图 16.9 归纳了位或操作符的运算结果。

| 位 1 | 位 2 | 位 1   位 2 |
|-----|-----|-----------|
| 0   | 0   | 0         |
| 1   | 0   | 1         |
| 0   | 1   | 1         |
| 1   | 1   | 1         |

图 16.9 利用位与操作符操作两个位的结果

常见编程错误 16.6 把位或操作符 `|` 和逻辑或操作符 `||` 混为一谈。

只有当两个操作数对应位仅有 1 位为 1 时,位异或<sup>①</sup>才将结果位设为 1。在图 16.7 中,变量 number1 与 number2 分别赋予值 139 (00000000 00000000 00000000 10001000) 和 199 (00000000 00000000 00000000 11000111)。当它们在表达式 number1 ^ number2 中进行操作时,可得结果 00000000 00000000 00000000 01001100。图 16.10 归纳了位异或操作符的运算结果。

| 位 1 | 位 2 | 位 1^位 2 |
|-----|-----|---------|
| 0   | 0   | 0       |
| 1   | 0   | 1       |
| 0   | 1   | 1       |
| 1   | 1   | 0       |

图 16.10 利用位或操作符操作两个位的结果

位取反 ~ 将操作数中为 1 的位对应的结果位置为 0,将操作数中为 0 的位对应的结果位置为 1,即以某种方式将取数的反码。计算表达式 ~number1 时,可得结果 (11111111 11111111 10101010 10101010)。

图 16.11 中的程序演示了左移操作符与右移操作符。函数 displayBits 用于打印无符号整型值。

```

1 //Fig.16.11: fig16_11.cpp
2 //Using the bitwise shift operators
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 void displayBits( unsigned );
14
15 int main()
16 {
17     unsigned number1 = 960;
18
19     cout << "The result of left shifting\n";
20     displayBits( number1 );
21     cout << "8 bit positions using the left "
22         << "shift operator is\n";
23     displayBits( number1 << 8 );
24     cout << "\nThe result of right shifting\n";
25     displayBits( number1 );
26     cout << "8 bit positions using the right "
27         << "shift operator is\n";

```

```

28     displayBits( number1 >> 8 );
29     return 0;
30 }
31
32 void displayBits( unsigned value )
33 {
34     const int SHIFT = 8 * sizeof( unsigned ) - 1;
35     const unsigned MASK = 1 << SHIFT;
36
37     cout << setw( 7 ) << value << " = ";
38
39     for ( unsigned c = 1; c <= SHIFT + 1; c++ ) {
40         cout << ( value & MASK ? '1' : '0' );
41         value <<= 1;
42
43         if ( c % 8 == 0 )
44             cout << " ";
45     }
46
47     cout << endl;
48 }

```

输出结果:

The result of left shifting

960 = 00000000 00000000 00000011 11000000

8 bit positions using the left shift operator is

245760 = 00000000 00000011 11000000 00000000

The result of right shifting

3 = 00000000 00000000 00000000 00000011

图 16.11 使用位取反操作符

左移操作符 << 将其左边指定的操作数的二进制表示向左移动右边指定的位数。右边空出的位用 0 来填充,而左边的移出的位则丢失。在图 16.11 中的程序中,变量 number1 被赋值是 960(00000000 00000000 00000011 11000000)。在表达式 number1 << 8 中 number1 左移 8 位的结果是 245 760(00000000 00000011 11000000 00000000)。

右移操作数 >> 将其左边的操作数的二进制表示向右移动其右边的指定位数。将一无符号整数右移会导致左边空出的位被 0 填充,而右边移出的位则丢失。在图 16.11 的程序中,在表达式 number1 >> 8 中 number1 右移 8 位的结果为 3(00000000 00000000 00000000 00000011)。

**常见编程错误 16.7** 如果移位操作符右边的数为负数或者大于左边操作数的实际位数,移位的结果将不确定。

**可移植性提示 16.4** 右移一个有符号值的结果与机器有关。有些机器用 0 填充,有些机器则用符号位来填充。

每一个位操作符(除了取反操作符外)都有其对应的赋值操作符,这些操作符如图

16.12 所示,它们与第 2 章介绍的代数赋值操作符用法类似。

| 位赋值操作符 |          |
|--------|----------|
| &=     | 位与赋值操作符  |
| =      | 位或赋值操作符  |
| ^=     | 位异或赋值操作符 |
| <<=    | 左移赋值操作符  |
| >>=    | 右移赋值操作符  |

图 16.12 位赋值操作符

图 16.13 简要介绍了不同操作符的优先级与结合性。

| 操作符                                                   | 结合性  | 类型    |
|-------------------------------------------------------|------|-------|
| ::(一元操作符,从右至左),::(二元操作符,从左至右)                         | 从左至右 | 最高级   |
| ( ), [ ], ., -> , ++ , -- , Static_cast <类型> ( )      | 从左至右 | 后缀表达式 |
| ++ , -- , + - , ! , delete , sizeof , * , & , new     | 从右至左 | 一元操作符 |
| *, / , %                                              | 从左至右 | 乘法类   |
| + -                                                   | 从左至右 | 加减类   |
| << , >>                                               | 从左至右 | 移位    |
| < , <= , > , >=                                       | 从左至右 | 关系操作符 |
| == , !=                                               | 从左至右 | 等式操作符 |
| &                                                     | 从左至右 | 位与    |
| ^                                                     | 从左至右 | 位异或   |
|                                                       | 从左至右 | 位或    |
| &&                                                    | 从左至右 | 逻辑与   |
|                                                       | 从左至右 | 逻辑或   |
| ?:                                                    | 从右至左 | 条件操作符 |
| = , += , -= , *= , /= , %= , &= ,  = , ^= , <<= , >>= | 从右至左 | 赋值操作符 |
| ,                                                     | 从左至右 | 逗号操作符 |

图 16.13 操作符优先级与结合性

16.8 位段

针对类或结构(或 union,参见第 18 章)中的 unsigned 或 int 成员,C++ 提供了为其指定存储位数的能力,这种成员被称为位段。利用位段可用最小的位数来存放数据以更好地利用内存。位段成员必须被声明为 unsigned 或 int 类型。

性能提示 16.2 位段有利于节省存储空间。

## 结构定义

```

struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};

```

包括了 3 个 unsigned 型位段: face, suit 和 color, 用于表示一副牌中的一张牌。声明位段的方式是这样的, 在 unsigned 或 int 成员后加一冒号, 然后再加一个表示位段宽度(即成员被存储的位数)的整型常量。位段宽度必须是在 0 和存储一个 int 型值所需的位数之间的一个整数常量。

前面的结构定义表明成员 face 在存储中占 4 位, 成员 suit 占 2 位, 成员 color 占 1 位。这些位数基于每个结构成员取值的范围。成员 face 存储在 0(表示牌 A)与 12(表示牌 K)之间的一个整数, 因此需要 4 位来存储(4 位可表示在 0 到 15 之间的任一个整数)。成员 suit 取值范围为 0 到 3(0 代表方块, 1 代表红桃, 2 代表梅花, 3 代表黑桃), 因此只需要 2 位来存储这个值。成员 color 要么取值为 0, 要么为 1, 因此只需 1 位来存储。

图 16.14 中的程序(输出结果如图 16.15 所示)创建了其中包含 52 个 BitCard 结构元素的一个数组。函数 fillDeck 将 52 张牌插入数组 deck, 函数 deal 打印这 52 张牌。注意, 结构的位段成员与其他结构成员访问方法是完全相同的。成员 color 用于在支持彩色显示的屏幕上显示一张牌的花色。

```

1 //Fig.16.14: fig16_14.cpp
2 //Example using a bit field
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 struct BitCard {
13     unsigned face : 4;    //4 bits; 0 - 15
14     unsigned suit : 2;    //2 bits; 0 - 3
15     unsigned color : 1;   //1 bit; 0 - 1
16 };
17
18 void fillDeck( BitCard * const );
19 void deal( const BitCard * const );
20
21 int main()
22 {
23     BitCard deck[ 52 ];
24
25     fillDeck( deck );
26     deal( deck );

```

```

27     return 0;
28 }
29
30 void fillDeck( BitCard * const wDeck )
31 {
32     for ( int i = 0; i <= 51; i++ ) {
33         wDeck[ i ].face = i % 13;
34         wDeck[ i ].suit = i / 13;
35         wDeck[ i ].color = i / 26;
36     }
37 }
38
39 //Output cards in two column format. Cards 0 -25 subscripted
40 //with k1 (column 1). Cards 26 -51 subscripted k2 in (column 2.)
41 void deal( const BitCard * const wDeck )
42 {
43     for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1 ++ , k2 ++ ) {
44         cout << "Card:" << setw( 3 ) << wDeck[ k1 ].face
45             << " Suit:" << setw( 2 ) << wDeck[ k1 ].suit
46             << " Color:" << setw( 2 ) << wDeck[ k1 ].color
47             << "    " << "Card:" << setw( 3 ) << wDeck[ k2 ].face
48             << " Suit:" << setw( 2 ) << wDeck[ k2 ].suit
49             << " Color:" << setw( 2 ) << wDeck[ k2 ].color
50             << endl;
51     }
52 }

```

图 16.14 利用位段保存一副扑克牌

输出结果:

```

Card: 0 Suit: 0 Color: 0 Card: 0 Suit: 2 Color: 1
Card: 1 Suit: 0 Color: 0 Card: 1 Suit: 2 Color: 1
Card: 2 Suit: 0 Color: 0 Card: 2 Suit: 2 Color: 1
Card: 3 Suit: 0 Color: 0 Card: 3 Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0 Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0 Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0 Card: 6 Suit: 2 Color: 1
Card: 7 Suit: 0 Color: 0 Card: 7 Suit: 2 Color: 1
Card: 8 Suit: 0 Color: 0 Card: 8 Suit: 2 Color: 1
Card: 9 Suit: 0 Color: 0 Card: 9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0 Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0 Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0 Card: 12 Suit: 2 Color: 1
Card: 0 Suit: 1 Color: 0 Card: 0 Suit: 3 Color: 1
Card: 1 Suit: 1 Color: 0 Card: 1 Suit: 3 Color: 1
Card: 2 Suit: 1 Color: 0 Card: 2 Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0 Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0 Card: 4 Suit: 3 Color: 1
Card: 5 Suit: 1 Color: 0 Card: 5 Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0 Card: 6 Suit: 3 Color: 1
Card: 7 Suit: 1 Color: 0 Card: 7 Suit: 3 Color: 1

```

```

Card: 8  Suit:1  Color:0  Card: 8  Suit:3  Color:1
Card: 9  Suit:1  Color:0  Card: 9  Suit:3  Color:1
Card:10  Suit:1  Color:0  Card:10  Suit:3  Color:1
Card:11  Suit:0  Color:0  Card:11  Suit:3  Color:1
Card:12  Suit:0  Color:0  Card:12  Suit:3  Color:1

```

图 16.15 图 16.14 所示程序的输出结果

C++ 中可以指定未命名的位段,此时位段在结构中用来作“填充项”。例如结构定义

```

Struct Example {
    unsigned a:13;
    unsigned :3;
    unsigned b:4;
};

```

中将 3 位的位段定义为“填充项”,其中不存放任何内容;成员 b 则存放于另一个存储单元中。宽度为 0 的未命名位段通常用于将下一个位段对齐于新的存储单元。例如结构定义

```

Struct Example {
    unsigned a:13;
    unsigned :0;
    unsigned b:4;
};

```

用一个未命名的 0 位位段来跳过 a 所在存储单元中剩余的位,以便在下一个存储单元中对齐成员 b。

**可移植性提示 16.5** 位段操作与机器有关。如有些机器允许位段跨越字边界,而有些机器则不允许这样做。

**常见编程错误 16.8** 试图访问位段中单独的位(像对待数组中的元素那样),位段不是“位的数组”。

**常见编程错误 16.9** 试图取位段的地址(操作符 & 不能用于位段,因为它们没有地址)。

**性能提示 16.3** 虽然位段能够节省空间,但它们也会导致编译器生成更慢的机器执行代码。这是因为访问一个存储单元中特定的部分需要额外的机器代码,是计算机科学中程序执行时间消耗与存储空间之间达到平衡的一个范例。

## 16.9 字符处理函数库

在计算机中,许多数据都是以字符的方式输入的,包括字母、数字和各种各样的特殊标识符。本节我们将讨论 C++ 的判断与操纵单个字符的能力。本章后续部分将继续讨论第 5 章开始介绍的字符串处理。

字符处理函数库包括了几个有用的进行字符判断与操纵的函数,每个函数都会将字符(被看作 int 或 EOF)作为其参数,字符通常按整数进行处理。在前面讲过,EOF 的值通常为 -1,而有些计算机硬件结构并不允许在 char 变量中存储负值,因此,字符处理函数把字符作为整数来处理。图 16.16 归纳了字符处理函数库中的函数,使用这些函数时,一定要包含头



文件 `<cctype>`。

| 函数原型                                | 描述                                                                                                                         |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>int isdigit ( int c )</code>  | 如果 <code>c</code> 为数字则返回 <code>true</code> , 否则返回 <code>false</code>                                                       |
| <code>int isalpha ( int c )</code>  | 如果 <code>c</code> 为字母则返回 <code>true</code> , 否则返回 <code>false</code>                                                       |
| <code>int isalnum ( int c )</code>  | 如果 <code>c</code> 为字母或数字则返回 <code>true</code> , 否则返回 <code>false</code>                                                    |
| <code>int isxdigit ( int c )</code> | 如果 <code>c</code> 为十六进制字符则返回 <code>true</code> , 否则返回 <code>false</code> (对二进制数、八进制数、十进制数、以及十六进制数的详细解释参见附录 C“数值系统”)        |
| <code>int islower ( int c )</code>  | 如果 <code>c</code> 为小写字母则返回 <code>true</code> , 否则返回 <code>false</code>                                                     |
| <code>int isupper ( int c )</code>  | 如果 <code>c</code> 为大写字母则返回 <code>true</code> , 否则返回 <code>false</code>                                                     |
| <code>int tolower ( int c )</code>  | 如 <code>c</code> 为大写字母, 则返回其小写形式; 否则返回未作改变的参数                                                                              |
| <code>int toupper ( int c )</code>  | 如 <code>c</code> 为小写字母, 则返回其大写形式; 否则返回未作改变的参数                                                                              |
| <code>int isspace ( int c )</code>  | 如果 <code>c</code> 为空格(“ ”)、进纸符(“\f”)、换行(“\n”)、回车(“r”)、横向跳格(“\t”)或纵向跳格(“\v”)则返回 <code>true</code> , 否则返回 <code>false</code> |
| <code>int iscntrl ( int c )</code>  | 如果 <code>c</code> 为控制符则返回 <code>true</code> , 否则返回 <code>false</code>                                                      |
| <code>int ispunct ( int c )</code>  | 如果 <code>c</code> 为不同于空格、数字或字母的打印字符, 则返回 <code>true</code> , 否则返回 <code>false</code>                                       |
| <code>int isprint ( int c )</code>  | 如果 <code>c</code> 为包括空格的打印字符则返回 <code>true</code> , 否则返回 <code>false</code>                                                |
| <code>int isgraph ( int c )</code>  | 如果 <code>c</code> 为不包括空格的打印字符则返回 <code>true</code> , 否则返回 <code>false</code>                                               |

图 16.16 字符处理函数库小结

图 16.17 演示了函数 `isdigit`, `isalpha`, `isalnum` 和 `isxdigit` 的功能。函数 `isdigit` 判断其参数是否为数字(0~9), 函数 `isalpha` 判断其参数是为大写字母(A~Z)还是小写字母(a~z), 函数 `isalnum` 判断其参数是大写字母、小写字母还是数字。函数 `isxdigit` 判断其参数是否为十六进制数(即 A~Z, a~f, 0~9)。

```

1 //Fig. 16.17: fig16_17.cpp
2 //Using functions isdigit, isalpha, isalnum and isxdigit
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cctype>
9
10 int main()
11 |
12     cout << "According to isdigit:\n"
13         << ( isdigit('8') ? "8 is a" : "8 is not a" )
14         << " digit\n"
15         << ( isdigit('#') ? "# is a" : "# is not a" )
16         << " digit\n";
17     cout << "\nAccording to isalpha:\n"
18         << ( isalpha('A') ? "A is a" : "A is not a" )
19         << " letter\n"
```

```

20     << ( isalpha('b') ? "b is a" : "b is not a" )
21     << " letter\n"
22     << ( isalpha('&') ? "& is a" : "& is not a" )
23     << " letter\n"
24     << ( isalpha('4') ? "4 is a" : "4 is not a" )
25     << " letter\n";
26     cout << "\nAccording to isalnum: \n"
27     << ( isalnum('A') ? "A is a" : "A is not a" )
28     << " digit or a letter\n"
29     << ( isalnum('8') ? "8 is a" : "8 is not a" )
30     << " digit or a letter\n"
31     << ( isalnum('#') ? "# is a" : "# is not a" )
32     << " digit or a letter\n";
33     cout << "\nAccording to isxdigit: \n"
34     << ( isxdigit('F') ? "F is a" : "F is not a" )
35     << " hexadecimal digit\n"
36     << ( isxdigit('J') ? "J is a" : "J is not a" )
37     << " hexadecimal digit\n"
38     << ( isxdigit('7') ? "7 is a" : "7 is not a" )
39     << " hexadecimal digit\n"
40     << ( isxdigit('$') ? "$ is a" : "$ is not a" )
41     << " hexadecimal digit\n"
42     << ( isxdigit('f') ? "f is a" : "f is not a" )
43     << " hexadecimal digit" << endl;
44     return 0;
45 }

```

#### 输出结果:

According to isdigit:

8 is a digit

# is not a digit

According to isalpha:

A is a letter

b is a letter

& is not a letter

4 is not a letter

According to isalnum:

A is a digit or a letter

8 is a digit or a letter

# is not a digit or a letter

According to isxdigit:

F is a hexadecimal digit

J is not a hexadecimal digit

7 is a hexadecimal digit

\$ is not a hexadecimal digit

f is a hexadecimal digit

图 16.17 isdigit, isalpha, isalnum 和 isxdigit 用法示例

图 16.17 中的程序使用条件操作符(?:),在打印字符输出结果时,确定是输出字符串"a"还是"s not a",如表达式

```
isdigit('8') ? "8 is a" : "8 is not a"
```

表明:如果'8'是一个数字(也就是说 isdigit 返回 true 值(即非 0)),则打印"8 is a",否则打印"8 is not a"。

图 16.18 中的程序演示了函数 islower, isupper, tolower 和 toupper 的功能。函数 islower 用于判断其参数是否为小写字母(A~Z),函数 isupper 判断其参数是否为大写字母(A~Z),函数 tolower 将一个大写字母转换为小写字母并返回这个小写字母,如果参数非大写字母,toupper 会原封不动返回其参数,函数 toupper 将一个小写字母转换为大写字母并返回这个大写字母。如果参数不是小写字母,toupper 会原封不动地返回其参数。

```
1 //Fig.16.18: fig16_18.cpp
2 //Using functions islower, isupper, tolower and toupper
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cctype>
9
10 int main()
11 {
12     cout << "According to islower: \n"
13         << ( islower('p') ? "p is a" : "p is not a" )
14         << " lowercase letter \n"
15         << ( islower('P') ? "P is a" : "P is not a" )
16         << " lowercase letter \n"
17         << ( islower('5') ? "5 is a" : "5 is not a" )
18         << " lowercase letter \n"
19         << ( islower('!') ? "! is a" : "! is not a" )
20         << " lowercase letter \n";
21     cout << "\nAccording to isupper: \n"
22         << ( isupper('D') ? "D is an" : "D is not an" )
23         << " uppercase letter \n"
24         << ( isupper('d') ? "d is an" : "d is not an" )
25         << " uppercase letter \n"
26         << ( isupper('8') ? "8 is an" : "8 is not an" )
27         << " uppercase letter \n"
28         << ( isupper('$') ? "$ is an" : "$ is not an" )
29         << " uppercase letter \n";
30     cout << "\nu converted to uppercase is "
31         << static_cast< char>( toupper('u') )
32         << "\n7 converted to uppercase is "
33         << static_cast< char>( toupper('7') )
34         << "\n $ converted to uppercase is "
35         << static_cast< char>( toupper('$') )
36         << "\nL converted to lowercase is "
37         << static_cast< char>( tolower('L') ) << endl;
```

```

38
39     return 0;
40 }

```

输出结果:

```

According to islower:
P is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l

```

图 16.18 islower, isupper, tolower 和 toupper 用法示例

图 16.19 中的程序演示了函数 isspace, iscntrl, ispunct, isprint 和 isgraph 的功能。函数 isspace 判断其参数是否为下列空白字符之一: 空格(" ")、进纸符("\f")、换行符("\n")、回车符("\r")、水平制表符("\t") 和垂直制表符("\v")。函数 iscntrl 判断其参数是否为下列控制符之一: 如水平制表符、垂直制表符、进纸符、警告符("\a")、退格符("\b")、回车符和换行符。函数 ispunct 判断其参数是否为以下不同于空格、数字或字母的打印字符之一: \$, #, (, ), [, ], {, }, ;, :, %, 等等。函数 isprint 用于判断其参数是否为能够在屏幕在显示的字符(包括空格)。除了在屏幕显示的字符不包括空格外函数 isgraph 的其他字符与 isprint 相同。

```

1 //Fig.16.19: fig16_19.cpp
2 //Using functions isspace, iscntrl, ispunct, isprint, isgraph
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cctype>
9
10 int main()
11 {
12     cout << "According to isspace:\nNewline "
13         << ( isspace('\n') ? "is a" : "is not a" )
14         << " whitespace character\nHorizontal tab "
15         << ( isspace('\t') ? "is a" : "is not a" )
16         << " whitespace character\n"
17         << ( isspace('%') ? "% is a" : "% is not a" )

```

```

18         << " whitespace character \n";
19
20     cout << " \nAccording to iscntrl: \nNewline "
21         << ( iscntrl('\n') ? "is a" : "is not a" )
22         << " control character \n"
23         << ( iscntrl('$') ? "$ is a" : "$ is not a" )
24         << " control character \n";
25
26     cout << " \nAccording to ispunct: \n"
27         << ( ispunct(';') ? "; is a" : "; is not a" )
28         << " punctuation character \n"
29         << ( ispunct('Y') ? "Y is a" : "Y is not a" )
30         << " punctuation character \n"
31         << ( ispunct('#') ? "# is a" : "# is not a" )
32         << " punctuation character \n";
33
34     cout << " \nAccording to isprint: \n"
35         << ( isprint('$') ? "$ is a" : "$ is not a" )
36         << " printing character \nAlert "
37         << ( isprint('\a') ? "is a" : "is not a" )
38         << " printing character \n";
39
40     cout << " \nAccording to isgraph: \n"
41         << ( isgraph('Q') ? "Q is a" : "Q is not a" )
42         << " printing character other than a space \nSpace "
43         << ( isgraph(' ') ? "is a" : "is not a" )
44         << " printing character other than a space" << endl;
45
46     return 0;
47 }

```

#### 输出结果:

According to isspace:  
 Newline is a whitespace character  
 Horizontal tab is a whitespace character  
 \$ is not a whitespace character

According to iscntrl:  
 Newline is a control character  
 \$ is not a control character  
 According to ispunct:  
 ; is a punctuation character  
 Y is not a punctuation character  
 # is a punctuation character

According to isprint:  
 \$ is a printing character  
 Alert is not a printing character

According to isgraph:  
 Q is a printing character other than a space

Space is not a printing character other than a space

图 16.19 使用 isspace, iscntrl, isprint 和 isgraph

## 16.10 字符串转换函数

第 5 章,我们讨论了 C++ 最常用的字符、字符串操作函数。随后几节里,我们要讨论其他函数,分别是:将字符串转换为数值的函数,查找字符串的函数以及操纵、比较、查找内存块的函数。

本章描述了通用函数库里的字符串转换函数,这些函数将数字字符串转换为整数和浮点数,图 16.20 列出了这些字符串转换函数。函数首部用 const 声明了变量 nPtr(参数内容从右至左可读作:“nPtr 是一个指向字符常量的指针”);const 表示参数值不能修改。使用通用函数库里的函数时,一定要包含头文件 <cstdlib>。

| 函数原型                                              | 说明                      |
|---------------------------------------------------|-------------------------|
| double atof( const char * nPtr)                   | 将字符 nPtr 转换为 double 值   |
| int atoi( const char * nPtr)                      | 将字符 nPtr 转换为 int 值      |
| long atol( const char * nPtr)                     | 将字符 nPtr 转换为 long int 值 |
| double strtod( const char * nPtr, char ** endPtr) | 将字符 nPtr 转换为 double 值   |

图 16.20 通用函数库里的字符串转换函数小结

函数 atof(图 16.21)将其参数(代表浮点数的字符串)转换为 double 值。如果字符串不能转换,比如字符串的第一个字符不是数字,函数就会返回 0。

```

1  /Fig.16.21: fig16_21.cpp
2  //Using atof
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstdlib>
9
10 int main()
11 {
12     double d = atof( "99.0" );
13
14     cout << "The string \"99.0\" converted to double is "
15          << d << " \nThe converted value divided by 2 is "
16          << d / 2.0 << endl;
17     return 0;
18 }
```

输出结果:

```

The string "90.0" converted to double is 99
The converted value divided by 2 is 49.5
```

图 16.21 函数 atof 用法示例

函数 `atoi`(图 16.22)将其参数(数字字符串)转换为 `int` 值,函数返回 `int` 值。如果字符串不能转换,函数将返回 0。

```

1 //Fig.16.22: fig16_22.cpp
2 //Using atoi
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstdlib>
9
10 int main()
11 {
12     int i = atoi( "2593" );
13
14     cout << "The string \"2593\" converted to int is " << i
15         << " \nThe converted value minus 593 is " << i - 593
16         << endl;
17     return 0;
18 }
```

输出结果:

```

The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

图 16.22 函数 `atoi` 用法示例

函数 `atol`(图 16.23)将其参数(代表 `long` 值的字符串)转换为长整型 `long` 值。如果字符串不能转换,函数将返回 0。如果 `int` 与 `long` 值都是以 4 个字节来存放的,函数 `atoi` 与函数 `atol` 的作用就会完全相同。

```

1 //Fig.16.23: fig16_23.cpp
2 //Using atol
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstdlib>
9
10 int main()
11 {
12     long x = atol( "1000000" );
13
14     cout << "The string \"1000000\" converted to long is " << x
15         << " \nThe converted value divided by 2 is " << x / 2
16         << endl;
17     return 0;
18 }
```

输出结果:

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

图 16.23 函数 atol 用法示例

函数 strtod(图 16.24)将表示浮点值的字符序列转换为 double 值。函数 strtod 接收 2 个参数:一个字符串(char \*)和一个字符串指针(也就是 char \*\*)。字符串包含了将转换为 double 值的字符序列,第二个参数赋值为字符串转换部分之后的第一个字符的位置。

```
1 //Fig.16.24: fig16_24.cpp
2 //Using strtod
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstdlib>
9
10 int main()
11 {
12     double d;
13     const char *string = "51.2% are admitted";
14     char *stringPtr;
15
16     d = strtod( string, &stringPtr );
17     cout << "The string \"" << string
18         << "\" is converted to the double value " << d
19         << " and the string \"" << stringPtr << "\" << endl;
20     return 0;
21 }
```

输出结果:

```
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
```

图 16.24 函数 strtod 用法示例

图 16.24 中的语句

```
d = strtod( string, &stringPtr );
```

表明 d 为转换后的字符串 string double 值,而 &stringPtr 为字符串中转换部分(51.2)之后第一个字符的位置。

函数 strtol(图 16.25)将表示整型值的字符序列转换为 long 值。函数 strtol 接收 3 个参数:一个字符串(char \*)、一个字符串指针(char \*\*)和一个整型值。字符串包含了要转换的字符序列,第二个参数赋值为字符串转换部分之后的第一个字符的位置,整数为要转换的值的基数。语句

```
x = strtol(string, &remainderPtr, 0);
```

表明 x 为转换后的字符串 string long 值,而 &remainderPtr 赋值为字符串中转换的部分之后第一个字符的位置。如果它为 0,则字符串转换后的剩余部分就会被忽略。第 3 个参数为 0 表明要转换的值可能是八进制、十进制或十六进制数。



```

1 //Fig.16.25: fig16_25.cpp
2 //Using strtol
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstdlib>
9
10 int main()
11 {
12     long x;
13     const char *string = "-1234567abc";
14     char *remainderPtr;
15
16     x = strtol( string, &remainderPtr, 0 );
17     cout << "The original string is \" << string
18         << "\"\n";
19     cout << "The converted value is " << x
20         << "\n";
21     cout << "The remainder of the original string is \""
22         << remainderPtr
23         << "\"\n";
24     cout << "The converted value plus 567 is "
25         << x + 567 << endl;
26     return 0;
27 }

```

输出结果:

```

The original string is "-1234567 abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000

```

图 16.25 函数 strtol 用法示例

在调用函数 strtol 时,基数可能是 0 或 2 到 36 之间的任意一个整数(对八进制、十进制、十六进制和二进制数值系统的详细的讨论参见附录 C“数值系统”)。基数为 11 到 36 之间的数字表示,用字母 A~Z 来表示数字 10 到 35。例如,十六进制的数可能包含数字 0~9 和字母 A~F。基数为 11 的整数可能包含数字 0~9 和字母 A,基数为 24 的整数可能包含数字 0~9 和字母 A~N,基数为 36 的整数可能包含数字 0~9 和字母 A~Z。

函数 strtoul(图 16.26)将表示 unsigned long 整数的字符串序列转换为 unsigned long 值,它的用法与 strtol 相同。图 16.26 所示程序中的语句

```
x = strtoul(string, &remainderPtr, 0);
```

表明 x 为转换后的字符串 string 的 unsigned long 值,而 &remainderPtr 赋值为字符串中转换部分之后第一个字符的位置。第 3 个参数为 0 表明要转换的值可能是八进制、十进制或十六进制数。

```

1 //Fig.16.26: fig16_26.cpp
2 //Using strtoul
3 #include <iostream>
4

```

```

5  using std::cout;
6  using std::endl;
7
8  #include <cstdlib>
9
10 int main()
11 {
12     unsigned long x;
13     const char *string = "1234567abc";
14     char *remainderPtr;
15
16     x = strtoul( string, &remainderPtr, 0 );
17     cout << "The original string is \" << string
18         << "\"\n";
19     cout << "The converted value is " << x
20         << "\n";
21     cout << "The remainder of the original string is \"
22         << remainderPtr
23         << "\"\n";
24     cout << "The converted value minus 567 is "
25         << x - 567 << endl;
26     return 0;
27 }

```

输出结果:

```

The original string is "1234567 abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000

```

图 16.26 函数 strtoul 用法示例

## 16.11 字符串处理函数库的查找函数

本节描述了字符串处理函数库中用于查找字符或字符串的函数。这些函数的小结如图 16.27 所示。注意函数 `strcspn` 和 `strspn` 指定的返回类型为 `size_t`, 类型 `size_t` 是标准中所定义的操作符 `sizeof` 返回值的整数类型。

| 函数原型                                                            | 描述                                                                                                                           |
|-----------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>char * strchr( const char * s, int c )</code>             | 定位字符串 <code>s</code> 首次出现字符 <code>c</code> 的位置。如果找到, 则返回指向 <code>c</code> 的指针; 否则返回空指针                                       |
| <code>char * strchr( const char * s, int c )</code>             | 定位字符串 <code>s</code> 最后一次出现字符 <code>c</code> 的位置。如果找到, 则返回指向 <code>c</code> 的指针; 否则返回空指针                                     |
| <code>size_t strspn( const char * s1, const char * s2 )</code>  | 确定并返回字符串 <code>s1</code> 中起始段的长度, 该起始段中只包含字符串 <code>s2</code> 中的字符                                                           |
| <code>char * strpbrk( const char * s1, const char * s2 )</code> | 定位字符串 <code>s1</code> 中首次出现字符串 <code>s2</code> 中任一字符的位置。如果找到 <code>s2</code> 中的一个字符, 则返回指向 <code>s1</code> 中这个字符的指针; 否则返回空指针 |
| <code>size_t strcspn( const char * s1, const char * s2 )</code> | 确定并返回字符串 <code>s1</code> 中不包含字符串 <code>s2</code> 中字符的起始段的长度                                                                  |
| <code>char * strstr( const char * s1, const char * s2 )</code>  | 定位字符串 <code>s1</code> 中首次出现字符串 <code>s2</code> 的位置。如果找到, 则返回指向 <code>s1</code> 中这个字符串的指针; 否则返回空指针                            |

图 16.27 查找字符串处理函数库

**可移植性提示 16.6** `size_t` 类型与机器有关,它代表 `unsigned long` 类型或 `unsigned int` 类型。

函数 `strchr` 在一个字符串查找一个字符首次出现的位置。如果找到, `strchr` 就返回指向字符串中这个字符的指针, 否则返回空。图 16.28 中的程序用 `strchr` 来查找“a”与“z”在字符串“This is a string”中首次出现的位置。

```

1 //Fig.16.28: fig16_28.cpp
2 //Using strchr
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     const char *string = "This is a test";
13     char character1 = 'a', character2 = 'z';
14
15     if ( strchr( string, character1 ) != NULL )
16         cout << '\\' << character1 << "'was found in \""
17             << string << "\".\n";
18     else
19         cout << '\\' << character1 << "'was not found in \""
20             << string << "\".\n";
21
22     if ( strchr( string, character2 ) != NULL )
23         cout << '\\' << character2 << "'was found in \""
24             << string << "\".\n";
25     else
26         cout << '\\' << character2 << "'was not found in \""
27             << string << "\"." << endl;
28     return 0;
29 }
```

输出结果:

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

图 16.28 函数 `strchr` 用法示例

函数 `strcspn`(图 16.29)确定其第一个参数(该参数不包含该函数第二个参数中的任何字符)起始段的长度,并返回此长度。

```

1 //Fig.16.29: fig16_29.cpp
2 //Using strcspn
3 #include <iostream>
4
5 using std::cout;
```

```

6  using std::endl;
7
8  #include <cstring>
9
10 int main()
11 {
12     const char *string1 = "The value is 3.14159";
13     const char *string2 = "1234567890";
14
15     cout << "string1 = " << string1 << "\nstring2 = " << string2
16         << "\n\nThe length of the initial segment of string1"
17         << "\n\ncontaining no characters from string2 = "
18         << strcspn( string1, string2 ) << endl;
19     return 0;
20 }

```

输出结果:

```

string 1 = The value is 3.14159
string 2 =1234567890

```

```

The length of the initial segment of string1
containing no characters from string2 =13

```

图 16.29 函数 `strcspn` 用法示例

函数 `strpbrk` 在其第一个字符串参数中查找首次出现第二个参数中任一字符的位置。如果找到第二个参数的任一字符, `strpbrk` 就返回指向第一个参数中这个字符的指针, 否则返回空。图 16.30 中的程序在 `string1` 中定位首次出现 `string2` 中任一字符的位置。

```

1  //Fig. 16.30: fig16_30.cpp
2  //Using strpbrk
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>
9
10 int main()
11 {
12     const char *string1 = "This is a test";
13     const char *string2 = "beware";
14
15     cout << "Of the characters in \" << string2 << "\"\n \""
16         << *strpbrk( string1, string2 ) << '\n'
17         << " is the first character to appear in\n \""
18         << string1 << '\n' << endl;
19     return 0;
20 }

```

输出结果:

```

Of the characters in "beware"

```

```
'a' is the first character to appear in
"This is a test"
```

图 16.30 函数 `strpbrk` 用法示例

函数 `strrchr` 在一个字符串查找最后一次出现指定字符的位置。如果找到,则返回指向字符串中该字符的指针,否则返回 0。图 16.31 中的程序在字符串“A zoo has many animals including zebras”中查找最后一次出现字符“z”的位置。

```
1 //Fig.16.31: fig16_31.cpp
2 //Using strrchr
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     const char *string1 = "A zoo has many animals "
13                           "including zebras";
14     int c = 'z';
15
16     cout << "The remainder of string1 beginning with the\n"
17           << "last occurrence of character '"
18           << static_cast< char>( c )
19           << "' is: \"" << strrchr( string1, c ) << "'\n" << endl;
20     return 0;
21 }
```

输出结果:

```
The remainder of string1 beginning with the
last occurrence of character 'z' is "zebras"
```

图 16.31 函数 `strrchr` 用法示例

函数 `strspn`(图 16.32)确定第一个字符串参数中只包含第二个字符串参数中字符的起始段的长度,并返回此长度。

```
1 //Fig.16.32: fig16_32.cpp
2 //Using strspn
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     const char *string1 = "The value is 3.14159";
13     const char *string2 = "aeihls Tuv";
```

```

14
15     cout << "string1 = " << string1
16         << " \nstring2 = " << string2
17         << " \n\nThe length of the initial segment of string1 \n"
18         << "containing only characters from string2 = "
19         << strspn( string1, string2 ) << endl;
20     return 0;
21 }

```

输出结果:

```

string1 = The value is 3.14159
string2 = aehils Tuv

```

```

The length of the initial segment of string1
containing only characters from string2 = 13

```

图 16.32 函数 `strspn` 用法示例

函数 `strstr` 在第一个字符串参数中查找首次出现其第二个字符串参数的位置。如果能在第一个字符串中找到第二个字符串,函数就返回第一个参数中相应位置的指针。图 16.33 中的程序利用 `strstr` 在字符串“abcdefabcdef”中查找字符串“def”。

```

1 //Fig.16.33: fig16_33.cpp
2 //Using strstr
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     const char *string1 = "abcdefabcdef";
13     const char *string2 = "def";
14
15     cout << "string1 = " << string1 << " \nstring2 = " << string2
16         << " \n\nThe remainder of string1 beginning with the \n"
17         << "first occurrence of string2 is; "
18         << strstr( string1, string2 ) << endl;
19     return 0;
20 }

```

输出结果:

```

string1 = abcdefabcdef
string2 = def

```

```

The remainder of string1 beginning with the
first occurrence of string2 is;defabcedef

```

图 16.33 函数 `strstr` 用法示例

## 16.12 字符串处理函数库中的内存处理函数

本节所提出的字符串处理函数有助于操作、比较和查找内存块。这些函数将内存块作为字符数组来处理。它们可以操作任何块数据。图 16.34 归纳了字符串处理函数库中的内存处理函数。讨论这些函数时,“对象”指代的是数据块。

| 函数原型                                                                     | 描述                                                                                                 |
|--------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>void * memcpy( void * s1, const void * s2, size_t n )</code>       | 从 s2 指向的对象中复制 n 个字符到 s1 指向的对象中。返回指向结果对象的指针                                                         |
| <code>void * memmove( void * s1, const void * s2, size_t n )</code>      | 从 s2 指向的对象中复制 n 个字符到 s1 指向的对象中。操作可按如下方式进行:首先将字符从 s1 指向的对象中复制到一个临时数组,然后将它们从临时数组复制到指向的对象。返回指向结果对象的指针 |
| <code>void * memcmp( const void * s1, const void * s2, size_t n )</code> | 比较 s1 与 s2 所指向对象的前 n 个字符。如果 s1 等于、小于或大于 s2,函数则分别返回 0、小于 0 或大于 0 的数                                 |
| <code>void * memchr( const void * s1, int c, size_t n )</code>           | 在 s 指向对象的前 n 个字符中定位首次出现 c(被转换为 unsigned char)的位置。如果找到返回对象中指向 c 的指针,否则返回 0                          |
| <code>void * memset( void * s, int c, size_t n )</code>                  | 将 c(被转换为 unsigned char)复制到 s 所指向对象的前 n 个字符中。返回指向结果的指针                                              |

图 16.34 字符串处理函数库中的内存处理函数

这些函数的指针参数都已声明为 `void *`。在第 5 章,我们看到指向任意数据类型的指针都可以直接赋给 `void *` 类型的指针。因此,这些函数能够接收任意数据类型的指针。记住, `void *` 类型的指针不能直接赋给任意一种数据类型的指针。因为这一特性,每一个函数要接收一个表示大小的参数来指定函数将要处理的字符(或字节)个数。为便于理解,本节中的例子仅对字符数组(字符块)进行处理。

函数 `memcpy` 从第一个参数指向的对象中把指定个数的字符(或字节)复制到第二个参数指向的对象中。它能接收任意类型对象的指针。如果两个参数所指向的对象重叠(也就是说,两者是同一对象的某一部分),函数的结果将不确定。图 16.35 中的程序利用 `memcpy` 将数组 `s2` 中的字符串复制到数组 `s1` 中。

```

1 //Fig.16.35: fig16_35.cpp
2 //Using memcpy
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char s1[ 17 ], s2[] = "Copy this string";
13
```

```

14     memcpy( s1, s2, 17 );
15     cout << "After s2 is copied into s1 with memcpy, \n"
16         << "s1 contains \"" << s1 << "\"\n" << endl;
17     return 0;
18 }

```

输出结果:

```

After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"

```

图 16.35 函数 memcpy 用法示例

函数 memmove 与 memcpy 类似,也是从第一个参数指向的对象中把指定个数的字符(或字节)复制到第二个参数指向的对象中。但其复制的过程可视为先将第二个参数中的相应数据复制到一个临时字符数组,然后再将临时数组中的数据复制到第一个参数。这类复制能将一个字符串的某一部分复制到同一个字符串的另一部分。

**常见编程错误 16.10** memmove 以外的字符串处理函数在同一字符串不同部分间相互复制时,会导致不确定的结果。

图 16.36 中的程序利用 memmove 将数组 x 的最后 10 个字节复制到你前 10 个字节。

```

1 //Fig.16.36: fig16_36.cpp
2 //Using memmove
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char x[] = "Home Sweet Home";
13
14     cout << "The string in array x before memmove is: " << x;
15     cout << "\nThe string in array x after memmove is: "
16         << static_cast<char * >( memmove( x, &x[ 5 ], 10 ) )
17         << endl;
18     return 0;
19 }

```

输出结果:

```

The string in array x before memmove is:Home Sweet Home
The string in array x memmove is:Sweet Home Home

```

图 16.36 函数 memmove 用法示例

函数 memcmp(如图 16.37)将第一个参数中指定个数的字符与第二个参数的相应字符进行比较。如果第一个参数比第二个大,函数就返回大于 0 的数值;相等则返回 0;如第一个参数比第二个小,函数就返回小于 0 的数值。

```

1 //Fig.16.37: fig16_37.cpp

```



```

2 //Using memcmp
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstring>
13
14 int main()
15 {
16     char s1[] = "ABCDEFGH", s2[] = "ABCDXYZ";
17
18     cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
19         << "\nmemcmp(s1, s2, 4) = " << setw( 3 )
20         << memcmp( s1, s2, 4 ) << "\nmemcmp(s1, s2, 7) = "
21         << setw( 3 ) << memcmp( s1, s2, 7 )
22         << "\nmemcmp(s2, s1, 7) = " << setw( 3 )
23         << memcmp( s2, s1, 7 ) << endl;
24     return 0;
25 }

```

输出结果:

```

s1 = ABCDEFGH
s2 = ABCDXYZ

```

```

memcmp(s1,s2,4) =  0
memcmp(s1,s2,7) = -1
memcmp(s2,s1,7) =  1

```

图 16.37 函数 memcmp 用法示例

函数 memchr 在一个对象的指定数目的字节中查找首次出现 unsigned char 类型字节的位置。如果找到,函数返回指向对象中这个字节的指针;否则返回空。图 16.38 中的程序在字符串“This is a string”中查找字符(字节)“r”。

```

1 //Fig.16.38: fig16_38.cpp
2 //Using memchr
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char s[] = "This is a string";
13

```

```

14     cout << "The remainder of s after character 'r'"
15         << "is found is \"
16         << static_cast<char * >( strchr( s, 'r', 16 ) )
17         << '\n' << endl;
18     return 0;
19 }

```

输出结果:

The remainder of s after character 'r' is found is "ring"

图 16.38 函数 `memchr` 用法示例

函数 `memset` 将第二个参数中的字节值复制到第一个参数指向的对象中指定数目的字节中。图 16.39 用 `memset` 将“b”复制到字符串 `string1` 中的前 7 个字节中。

```

1 //Fig.16.39: fig16_39.cpp
2 //Using memset
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     char string1[ 15 ] = "BBBBBBBBBBBBBB";
13
14     cout << "string1 = " << string1 << endl;
15     cout << "string1 after memset = "
16         << static_cast<char * >( memset( string1, 'b', 7 ) )
17         << endl;
18     return 0;
19 }

```

输出结果:

```

string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB

```

图 16.39 函数 `memset` 用法示例

## 16.13 字符串处理函数库中的其他函数

字符串处理函数库中还有一个函数为 `strerror`, 图 16.40 描述了它的用法。函数 `strerror` 将一个错误编号作为参数, 创建一条错误消息字符串。函数返回指向这个字符串的指针。图 16.41 中的程序演示了函数 `strerror` 的用法。

| 函数原型                                       | 描述                                                      |
|--------------------------------------------|---------------------------------------------------------|
| <code>char * strerror( int errnum )</code> | 以系统相关方式, 把 <code>errnum</code> 映射为完整的文本字符串, 返回指向该字符串的指针 |

图 16.40 字符串处理函数库中的另一个字符串处理函数

```
1 //Fig.16.41: fig16_41.cpp
2 //Using strerror
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>
9
10 int main()
11 {
12     cout << strerror( 2 ) << endl;
13     return 0;
14 }
```

输出结果:

No such file or directory

图 16.41 函数 `strerror` 用法示例

可移植性提示 16.7 `strerror` 生成的消息与系统有关。

## 16.14 小结

- 结构是采用同一个名字的相关变量的集合,有时也被称为联合体。
- 结构可以包含不同数据类型的变量。
- 每个结构定义均以关键字 `struct` 开头。包含在定义花括号内是结构成员的声明。
- 同一结构的成员名必须惟一。
- 结构定义创建了一种新的数据类型,可用于声明变量。
- 结构能够用初始化数据列表来初始化。要想初始化一个结构,可在其结构变量声明后加一等号,然后放置用花括号括起来的初始化数据。如果初始化数据列表少于成员数,余下的成员就会初始化为 0(对指针成员来说为空)。
- 所有的结构变量都可以赋值给同类型的另一个结构变量。
- 一个结构变量可以初始化为同类型的另一个结构变量。
- 结构变量和结构成员是按值传递给函数的,数组成员则是按引用传递的。
- 要想按引用传递结构,可传递结构变量的地址,结构数组是按引用来传递的。要想按值传递数组,可创建将数组作为成员的结构。
- 用 `typedef` 创建的新类型名并不含创建新的数据类型。它只是创建了已定义类型的同义名或别名。
- 位与操作符 `&` 对两个整型操作数进行操作。各操作数的对应位均为 1 时结果位才为 1。
- 掩码在保存某些位时用于隐藏其他位。
- 位或操作符 `|` 对两个整型操作数进行操作。两个操作数的对应位中只要有一个为

1,结果位就为1。

- 位操作符(除位取反外)都有其相应的赋值操作符。
- 位异或“^”对两个整型操作数进行操作。两个操作数对应位中只有一个为1时,结果位才为1。
- 位左移操作符“<<”将其左边的操作数向左移动右边所指定的位数。右边空出的位用0填充。
- 位右移操作符“>>”将其左边的操作数向右移动其右边指定的位数。对无符号整数进行右移操作会在左边空出的位中填充0。对有符号的整数进行右移操作则可能会填充符号位,这与机器有关。
- 位取反操作符“~”对一个操作数进行操作并反转操作数的各位,结果为操作数的反码。
- 位段通过以最小位数来存放数据以节省存储空间。位段成员必须声明为 int 或 unsigned 类型。
- 位段宽度必须是一个整数常量,而且必须位于在0和机器中用于存储 int 变量所需的位数之间。
- 如果指定位段未命名,那么它会用作结构中的填充项。
- 未命名的0宽度的位段用将下一个位段的新的机器字边距存放。
- 函数 islower 判断其参数是否为小写字母(a~z),函数 isupper 判断其参数是否为大写字母(A~Z)。
- 函数 isdigit 判断其参数是否为数字(0~9)。
- 函数 isalpha 判断其参数是否为大写字母(A~Z)或小写字母(a~z)。
- 函数 isalnum 判断其参数是否为大写字母、小写字母或数字。
- 函数 isxdigit 判断其参数是否为十六进制的数字(即 A~Z, a~f, 0~9)。
- 函数 tolower 将一个大写字母转换为小写字母并返回这个小写字母,函数 toupper 将一个小写字母转换为大写字母并返回这个大写字母。
- 函数 isspace 判断其参数是否为下列空白符之一,空格(“ ”)、进纸符(“\f”)、换行符(“\n”)、回车符(“\r”)、水平制表符(“\t”)或垂直制表符(“\v”)。
- 函数 iscntrl 判断其参数是否为下列控制符之一水平制表符、垂直制表符、进纸符、响铃符(“\a”)、退格符(“\b”)、回车或者换行符。
- 函数 ispunct 判断其参数是否为不同于空格、数字或字母的可打印字符: \$, #, (, ), [, ], {, }, ;, :, %, 等等。
- 函数 isprint 判断其参数是否为可打印字符(包括空格)。
- 函数 isgraph 判断其参数是否为可打印字符(不包括空格)。
- 函数 atof(图 16.21)将其参数(代表浮点数的字符串)转换为 double 值。
- 函数 atoi(图 16.22)将其参数(数字字符串)转换为 int 值。函数返回 int 值。
- 函数 atol(图 16.23)将其参数(代表 long 值的字符串)转换为 long 值。
- 函数 strtod(图 16.24)将表示浮点值的字符序列转换为 double 值。函数 strtod 接收2个参数:一个 char \* 类型的字符与一个指向 char \* 数据类型的指针。字符串包含了

即将转换为 double 值的字符序列,字符串转换之后的其余部分则赋给指向 char \* 数据类型的指针。

- 函数 `strtol` (图 16. 25) 将表示整型值的字符序列转换为 long 值。函数 `strtol` 接收 3 个参数: 一个字符 (char \*), 一个指向 char \* 的指针和一个整型值。字符串包含了即将转换的字符序列, 字符串转换之后的其余部分赋给指向 char \* 数据类型的指针, 整数为即将转换的值的基数。
- 函数 `strtoul` (图 16. 25) 将表示整型值的字符序列转换为 unsigned long 值。函数 `strtoul` 接收 3 个参数: 一个字符串 (char \*), 一个指向 char \* 的指针和一个整型值。字符串包含了即将转换的字符序列, 字符串转换部分之后的其余部分赋给指向 char \* 数据类型的指针。整数为即将转换的值的基数。
- 函数 `strchr` 在一个字符串查找首次出现指定字符的位置。如果字符找到, `strchr` 就返回字符串中指向该字符的指针, 否则返回空。
- 函数 `strcspn` (图 16. 29) 计算第一个参数中不包含第二个参数任何字符的起始段的长度, 并返回该长度。
- 函数 `strpbrk` 在第一个字符串参数中查找首次出现第二个参数中任一字符的位置。如果找到第二个参数的任一字符, `strpbrk` 则返回第一个参数中指向该字符的指针, 否则返回空。
- 函数 `strrchr` 在一个字符串查找最后一次出现指定字符的。如果找到, 则返回字符串中指向该字符的指针, 否则返回空。
- 函数 `strspn` (图 16. 32) 计算其第一个字符串参数中包含第二个字符串参数中任一字符的起始段的长度, 并返回该长度。
- 函数 `strstr` 在第一个字符串参数中查找首次出现第二个字符串参数的位置。如果能够在第一个字符串中找到第二个字符串, 函数返回指向第一个参数中相应位置的指针。
- 函数 `memcpy` 从第一个参数指定的对象中把一定数目的字符 (或字节) 复制到第二个参数指定的对象。它能接收任意类型对象的指针。memcpy 用 void \* 指针接收指向任何一种数据类型的指针, 并在使用时转换为 char \*。操作时, 函数将参数中的字节视为字符。
- 函数 `memmove` 从第一个参数指定的对象中把一定个数的字符 (或字节) 复制到第二个参数指定的对象中。但其复制的过程可视为先将第二个参数中的相应数据复制到一个临时字符数组, 然后再将临时数组中的数据复制到第一个参数。
- 函数 `memcmp` (如图 16. 37) 将它的第一个参数中指定位数的字符与第二个参数的相应字符进行比较。
- 函数 `memchr` 在一个对象指定数目的字节中查找首次出现一个表示无符号 char 数据的字节的位置。如果找到, 函数返回指向对象中这个字节的指针; 否则返回空。
- 函数 `memset` 将第二个参数中的字节值复制到第一个参数所指向的对象中一定数目的字节中。
- 函数 `strerror` 取错误编号作为参数, 并产生一条错误消息字符串。函数将返回指向该字符串的指针。

## 本章术语

arrays of structures 结构数组

$\wedge$  = bitwise exclusive - OR assignment operator

位异或赋值操作符 $\wedge$ =

$\wedge$  bitwise exclusive - OR operator 位异或操作符 $\wedge$

$|$  = bitwise inclusive - OR assignment operator

位或赋值操作符 $|$ =

bit field 位段

$\&$  = bitwise AND assignment operator

位与赋值操作符 $\&$ =

bitwise AND operator 位与操作符 $\&\&$

$|$  bitwise inclusive - OR operator 位或操作符 $|$

bitwise operators 位操作符

character code 字符编码

character const 字符常量

character set 字符集

complementing 取反

control character 控制符

delimiter 分隔符

general utilities library 通用函数库

hexadecimal digits 十六进制数字

initialization of structures 结构初始化

$\<=$  left - shift assignment operator

左移赋值操作符 $\<=$

$\<\<$  left - shift operator 左移操作符 $\<\<$

leftshift 左移

literal 直接量

masking off bits 位屏蔽

mask 掩码

one's complement 反码

one's - complement operator 取反操作符 $\sim$

padding 填充

pointer to a structure 结构指针

printing character 打印字符

record 记录

$\>=$  right - shift assignment operator

右移赋值操作符 $\>=$

right shift 右移位

$\>\>$  right - shift operator 右移操作符 $\>\>$

search string 查找字符串

self - referential structure 自引用结构

shifting 移位

space - time trade - offs 空间 - 时间冲突

string constant 字符常量

string conversion functions 字符转换函数

string literals 字符串直接量

string processing 字符处理

string 字符串

struct assignment 结构赋值

structure initialization 结构初始化

structure type 结构类型

unnamed bit field 未命名位段

white - space characters 空白字符

width of bit field 位段宽度

word - processing 字处理

zero - width bit field 零宽度位段

## 常见编程错误

16.1 结构定义结束处忘记加分号。

16.2 比较结构会导致语法错误,因为不同系统的边界对齐要求不同。

16.3 认为结构像数组那样按引用传递的,并试图在被调用函数中修改调用函数中结构的值。

16.4 引用结构数组中的单个结构时,忘记包含数组下标。

16.5 将位与操作符 $\&$ 错用作逻辑与操作符 $\&\&$ ,或者反之。

16.6 将位或操作符 $|$ 错用作逻辑或操作符 $||$ ,或者反之。

16.7 如果移位操作符右边的数为负数或大于左边操作数的实际位数,移位的结果将不确定。

- 16.8 试图访问位段中的单独的位(就像对待数组中的元素那样)。位段不是“位的数组”。
- 16.9 试图取位段的地址(操作符 & 不能用于位段,因为它们没有地址)。
- 16.10 memmove 以外的字符串处理函数在同一字符串的不同部分相互间复制时,会导致不确定的结果。

### 良好编程习惯

- 16.1 创建结构时始终提供结构名。这样一来,在程序后面定义这种结构类型的变量时就会很方便。将结构作为参数传递给函数时,结构名是必须的。
- 16.2 大写用 typedef 定义的新类型名以表明它们是其他类型名的同义名或别名。用 typedef 定义一种新的类型名并没有创建一种新的数据类型。typedef 只是创建了一个新的类型名,它可以被用来作为已有类型的别名。

### 性能提示

- 16.1 按引用传递结构比按值传递结构(这需要复制整个结构)更有效率,尤其是大型结构。
- 16.2 位段有利于节省存储空间。
- 16.3 虽然位段能够节省空间,但它们也会导致编译器生成更慢的机器执行代码。这是因为访问一个存储单元中特定的部分需要额外的机器代码。这是计算机科学中程序执行时间消耗与存储空间之间达到平衡的一个例子。

### 可移植性提示

- 16.1 因为一种特定数据类型的大小与机器有关以及存储边界对齐也与机器有关,所以结构的数据表示法也与机器有关。
- 16.2 用 typedef 可增强程序可移植性。
- 16.3 位数据操与机器硬件有关。
- 16.4 右移一个有符号值的结果与机器有关。有些机器用 0 填充而有些机器用符号位来填充。
- 16.5 位段操作与机器有关。如有些机器允许位段跨越字边界,而有些机器则不允许。
- 16.6 类型 size\_t 与机器有关,它代表类型 unsigned long 或类型 unsigned int。
- 16.7 strerror 创建的消息与系统有关。

### 自测题

#### 16.1 填空题:

- a) \_\_\_\_\_是具有相同名称的相关变量的集合。
- b) 使用\_\_\_\_\_操作符在两个操作数对应位皆为 1 时将结果位置为 1,否则将结果位置为 0。
- c) 在结构定义中声明的变量被称为结构的\_\_\_\_\_。
- d) 使用\_\_\_\_\_操作符在两个操作数对应位只要有一位为 1 时,就将结果位置为 1,否则将结果位置为 0。

- e) 使用关键字 \_\_\_\_\_ 进行结构定义。
- f) 使用关键字 \_\_\_\_\_ 创建已定义数据类型的同义名或别名。
- g) 使用 \_\_\_\_\_ 操作符在两个操作数对应位仅有一位为1时就将结果位置为1,否则将结果位置为0。
- h) 位与操作符 & 用来 \_\_\_\_\_ 位(也就是从位串中选择特定的位,并将其他置0)。
- i) 结构名被称为结构的 \_\_\_\_\_。
- j) 结构成员用操作符 \_\_\_\_\_ 或 \_\_\_\_\_ 来访问。
- k) 操作符 \_\_\_\_\_ 和 \_\_\_\_\_ 分别用于将一个值左移或者右移若干位。

#### 16.2 判断正误,如有错,请说明原因。

- a) 结构只能包含一种数据类型。
- b) 不同结构的成员,必须有惟一的名称。
- c) 关键字 typedef 用于定义新的数据类型。
- d) 结构始终按引用传递给函数。

#### 16.3 编写一条或一组语句,完成下列操作:

- a) 定义结构 Part,其成员包括一个 int 变量 partNumber 和一个拥有 25 个元素的 char 数组 partName。
- b) 将 partPtr 定义为类型 Part \* 的同义名。
- c) 声明 Part 类型的变量 a,数组 b[10]以及指针变量 ptr。
- d) 从键盘输入变量 a 的成员值。
- e) 将变量 a 赋给数组 b[10]的第3个元素。
- f) 将数组 b[10]的地址赋给指针变量 ptr。
- g) 用变量 ptr 和结构指针成员操作符来打印数组 b[10]中第3个元素的成员值。

#### 16.4 指出下列语句中的错误。

- a) 假定结构 Card 包含两个 char 类型的指针成员,即 face 和 suit。变量 c 的声明类型为 Card,变量 cPtr 的声明类型为指向 Card 的指针。变量 cPtr 的赋值为 c 的地址。则

```
cout << *cPtr.face << endl;
```

- b) 假定结构 Card 包含两个类型为 char 的指针成员:face 和 suit。数组 hearts[13]声明为类型 Card。语句

```
cout << hearts.face << endl;
```

- c) 要打印数组第10个元素的成员 face 的值。

```
struct Person{
    char lastName[15];
    char firstName[15];
    int age;
```

- d) 假定变量 p 声明为类型 Person,并且变量 c 声明为类型 Card 那么

```
p = c
```

#### 16.5 编写语句完成下列操作。假定变量 c(存储一个字符),x,y 和 z 的类型为 int。变量 d,e 和 f 的类型为 double;变量 ptr 的类型为 char \*,数组 s1[100]与 s2[100]的类型为



char。

- a) 将变量 c 中的字符转换为大写字母,然后将结果赋值给变量 c。
- b) 确定变量 c 是否为数字。打印数据时使用如图 16.17,16.18 和 16.19 中的条件操作符确定打印“is a”还是打印“is not a”。
- c) 将字符串“1234567”转换为 long 值并打印。
- d) 确定变量 c 是否为控制字符。打印数据时使用条件操作符来确定是打印“is a”还是打印“is not a”。
- e) 将在 s1 中 c 的最后一次出现的位置赋值给变量 ptr。
- f) 将字符串“8.63582”转换为 double 值,并打印。
- g) 确定变量 c 是否为字母。打印数据时使用条件操作符确定打印“is a”还是打印“is not a”。
- h) 将首次出现 s2 在 s1 中的位置赋值给 ptr。
- i) 确定变量 c 是否为打印字符。打印数据时用条件操作符确定打印“is a”还是打印“is not a”。
- j) 将 s2 任意一个字符在 s1 中首次出现的位置赋值给 ptr。
- k) 将 c 在 s1 中的首次出现的位置赋值给 ptr。
- l) 将字符串“-21”转换为 int,并打印。

### 自测题答案

- 16.1 a) 结构 b) 位与操作符 & c) 成员 d) 位或操作符(|) e) struct f) typedef  
g) 位异或操作符(^) h) 掩码 i) 标记符 j) 结构成员(.),结构指针(->) k)  
左移操作符(<<),右移操作符(>>)
- 16.2 a) 错误。结构可以包含多种数据类型。  
b) 错误。不同结构的成员名字可以相同,但同一结构的成员名字不能相同。  
c) 错误。typedef 用于定义已定义数据类型的别名。  
e) 错误。结构始终按值传递。
- 16.3 a) 

```
struct Part{
    int partNumber;
    char partName[26];
};
```

  
b) 

```
typedef Part * PartPtr;
```

  
c) 

```
Part a, b[10], *ptr;
```

  
d) 

```
cin >> a.partNumber >> a.partName;
```

  
e) 

```
b[3] = a;
```

  
f) 

```
ptr = b;
```

  
g) 

```
cout << (ptr + 3) -> partNumber << "
    << (ptr + 3) -> partName << endl;
```
- 16.4 a) 错误:应用括号将 \*cPtr 括起来。否则表达式的计算次序就不正确。  
b) 错误:忽略了数组下标。表达式应为 hearts[10].face。

- c) 错误:结构定义应以分号结尾。
- d) 错误:不同类型的结构变量不能相互赋值。

16.5 a) `c=toupper(c);`  
 b) `cout << '\ ' << c << " \ "`  
     `<<(isdigit(c)?"is a":"is not a")`  
     `<< " digit" << endl;`  
 c) `cout << atol("1234567") << endl;`  
 d) `cout << '\ ' << c << " \ "`  
     `<<(isctrl(c)?"is a":"is not a")`  
     `<< " control character" << endl;`  
 e) `ptr=strrchr(s1,c);`  
 f) `out << atof("8.63582") << endl;`  
 g) `cout << '\ ' << c << " \ "`  
     `<<(isalpha(c)?"is a":"is not a")`  
     `<< " letter" << endl;`  
 h) `ptr=strstr(s1,s2);`  
 i) `cout << '\ ' << c << " \ "`  
     `<<(isprint(c)?"is a":"is not a")`  
     `<< " printing character" << endl;`  
 j) `ptr=strpbrk(s1,s2);`  
 k) `ptr=strchr(s1,c);`  
 l) `cout << atoi("-21") << endl;`

### 练习题

16.6 指出下列结构与联合体的定义。

- a) 结构 Inventory, 其成员包括字符数组 `partName[30]`, 整型数 `partNumber`, 浮点数 `price`, 整型数 `stock` 和整型数 `recorder`。
- b) 结构 Address, 其成员包括字符数组 `streetAddress[25]`, `city[20]`, `state[3]` 和 `zipCode[6]`。
- c) 结构 Student, 其成员包括数组 `firstName[15]` 和 `lastName[15]`, 类型为 b) 中结构 Address 的变量 `homeAddress`。
- d) 结构 Test, 包括 16 个宽度为 1 的位段。位段名从字母 a~p。

16.7 根据以下结构定义与变量声明, 编写分别用于访问下列数据成员的表达式。

```
struct Customer{
    char lastName[15];
    char firstName[15];
    int customerNumber;
    struct{
        char phoneNumber[11];
        char address[50];
        char City[15];
        char State[3];
```

```

        char zipC[6];
    }personal;
}customerRecord, *customerPtr;

```

```
customerPtr = &customerRecord;
```

分别编写用于访问下列结构成员的表达式:

- a) 结构 customerRecord 的成员 lastName;
- b) customerPtr 所指向的结构的成员 lastName;
- c) 结构 customerRecord 的成员 firstName;
- d) customerPtr 所指向的结构的成员 firstName;
- e) 结构 customerRecord 的成员 customerNumber;
- f) customerPtr 所指向的结构的成员 customerNumber;
- g) 结构 customerRecord 的成员 personal 的成员 phoneNumber;
- h) customerPtr 所指向的结构的成员 personal 的成员 phoneNumber;
- i) 结构 customerRecord 的成员 personal 的成员 address;
- j) customerPtr 所指向的结构的成员 personal 的成员 address;
- k) 结构 customerRecord 的成员 personal 的成员 city;
- l) customerPtr 所指向的结构的成员 personal 的成员 city;
- m) 结构 customerRecord 的成员 personal 的成员 state;
- n) customerPtr 所指向的结构的成员 personal 的成员 state;
- o) 结构 customerRecord 的成员 personal 的成员 zipCode;
- p) customerPtr 所指向的结构的成员 personal 的成员 zipCode。

- 16.8 修改图 16.14 中的程序,并运用图 16.2 中所示的高性能洗牌算法来洗牌。以图 16.3 所示的两列格式打印这副牌,并在每张牌前标注其颜色。
- 16.9 编写程序将一个整型变量右移 4 位,程序应按位打印操作前后的整型变量。回答问题:你的机器在左边空出的位中填充的是 0 还是 1 呢?
- 16.10 如果你的计算机使用的整数占 4 字节,那么修改图 16.5 中的程序以便它能处理 4 字节整数。
- 16.11 将一个无符号整数左移一位相当于把这个整数乘以 2。编写接收两个参数 number 与 pow 的函数 power2,并计算  $\text{number} * 2^{\text{pow}}$ 。利用移位操作符来计算结果,程序应按位打印整数。
- 16.12 左移操作符能用于将两个字符打包成一个 2 个字节的无符号整数变量。编写程序从键盘输入两个字符并将它们传递给函数 packCharacters。要想将两个字符打包成一个 unsigned 整型变量,先将第一个字符赋给一个 unsigned 变量,然后将它左移 8 位并用位或操作符(&)将它与第二个字符连接起来。程序应按位输出打包前后的字符以表明打包操作无误。
- 16.13 利用右移操作符、位与操作符和掩码,编写函数 unpackCharacters 将练习 16.12 中的 unsigned 整数解成两个字符。要达到此目的,可将这个 unsigned 整数与掩码 65280 (11111111 00000000)相与然后将结果右移 8 位,再将结果赋给一个字符变量。接下来将这个 unsigned 整数与掩码 255 (00000000 11111111)相与,并将结果赋给另一个

整型变量。程序应按位输出分解前后的字符以表明分解操作无误。

- 16.14 如果机器使用的整数占 4 个字节,那么改写练习题 16.12 中的程序来操作 4 个字符。
- 16.15 如果机器使用的整数占 4 个字节,改写练习 16.13 中的函数 `unpackCharacters` 将一个整型变量分解包成 4 个字符。通过将掩码变量中的值 255 向左移动 8 位 0~3 次(与分解的字节有关)来获得分解包所需的掩码。
- 16.16 编写程序将一个无符号整数值的二进制位反序排列。程序应由用户输入一个整数值,然后调用函数 `reverseBits` 反序打印该整数中的各位。程序应按位打印操作前后的整数以确保操作成功。
- 16.17 编写一个程序,演示如何按值传递数组(提示:使用结构)。并在被调用的函数中修改数组的副本来证实传递的的确为数组的副本。
- 16.18 编写程序,从键盘输入一个字符并用字符处理函数库中的每个函数来测试该字符。程序应打印各函数返回的值。
- 16.19 下面的程序用函数 `multiple` 来确定从键盘输入的一个整数是否是某个整数  $X$  的倍数。分析函数 `multiple` 并确定  $X$  的值。

```

1 //ex16_19.cpp
2 //This program determines if a value is a multiple of X
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 bool multiple( int );
10
11 int main()
12 {
13     int y;
14
15     cout << "Enter an integer between 1 and 32000: ";
16     cin >> y;
17
18     if ( multiple( y ) )
19         cout << y << " is a multiple of X" << endl;
20     else
21         cout << y << " is not a multiple of X" << endl;
22
23     return 0;
24 }
25
26 bool multiple( int num )
27 {
28     bool mult = true;
29
30     for ( int i = 0, mask = 1; i < 10; i++ , mask <<= 1 )
31         if ( ( num & mask ) != 0 ) {
32             mult = false;

```

```
33         break;
34         |
35
36         return mult;
37     |
```

16.20 指出下列程序的用途。

```
1  //ex16_20.cpp
2  #include <iostream>
3
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int mystery( unsigned );
9
10 int main()
11 |
12     unsigned x;
13
14     cout << "Enter an integer: ";
15     cin >> x;
16     cout << "The result is " << mystery( x ) << endl;
17     return 0;
18 |
19
20 int mystery( unsigned bits )
21 |
22     const int SHIFT = 8 * sizeof( unsigned ) - 1;
23     const unsigned MASK = 1 << SHIFT;
24     unsigned total = 0;
25
26     for ( int i = 0; i < SHIFT + 1; i++, bits <<= 1 )
27         if ( ( bits & MASK ) == MASK )
28             ++total;
29
30     return ! ( total % 2 );
31 |
```

- 16.21 编写程序,用 `istream` 的成员函数 `getline`(参见第 11 章)把一行文本输入到字符数组 `s[100]`。用大写和小写字母输出这行文本。
- 16.22 编写程序,输入 4 个代表整数数值的字符串,然后将它们转换为整数并计算它们的和,最后打印出来。
- 16.23 编写程序,输入 4 个代表浮点数值的字符串,然后将它们转换为 `double` 数并计算它们的和,最后打印出来。
- 16.24 编写程序,从键盘输入一行文本和一个要查找的字符串。利用函数 `strstr` 在这行文本中定位首次出现查找字符串的位置,并将此位置赋给 `char *` 类型的变量 `searchPtr`。如果找到要查找字符串,打印该文本中该字符串之后的部分。如果第二次能找到该字

符串出现,打印该行文本中第二次出现查找字符串以后的部分(提示:第二次调用 `strstr` 时将 `searchPtr + 1` 作为其第一个参数)。

- 16.25 编写基于练习题 16.24 的程序输入几行文本和一个查找字符串,用函数 `strstr` 确定查找字符串在所输入文本中出现的次数并打印结果。
- 16.26 编写程序输入几行文本和一个查找字符串,用函数 `strstr` 确定查找字符串在所输入文本中出现的总次数。
- 16.27 编写基于练习题 16.26 的程序输入几行文本并确定字母表中每一个字母在所输入文本中出现的次数。不分大小写。将每个字母出现的次数存储在一个数组里,确定所有字母出现的次数后以表格的形式打印这些次数。
- 16.28 附录 B 列出了 ASCII 字符集中字符的数字代码。仔细研究此表,判断下列表达式是否正确:
  - a) 字母“A”出现在“B”之前;
  - b) 数字“9”出现在“0”之前;
  - c) 常用的加、减、乘与除符号出现在所有数字之前;
  - d) 数字出现在字母之前;
  - e) 如果一个排序程序按升序排序字符串,那么它会把右花括号放在左花括号之前。
- 16.29 编写程序读入一系列字符串,只打印以字母“b”开头的字符串。
- 16.30 编写程序读入一系列字符串,只打印以字母“ED”结尾的字符串。
- 16.31 编写程序输入一个 ASCII 码并打印相应的字符。修改程序使之产生 000 和 255 之间所有的三位数字的数码并打印相应的字符。说说程序运行时会出现什么结果?
- 16.32 参考附录 B 中的 ASCII 字符表,编写图 16.16 中的字符处理函数。
- 16.33 编写图 16.20 中的函数,将字符串转换为数字。
- 16.34 编写图 16.27 中的函数,查找字符串。
- 16.35 编写图 16.34 中的函数,操作内存块。
- 16.36 (项目:拼写检查器)很多流行的字处理软件包都内置拼写检查器。在编写本书时我们就用到了拼写检查的功能。同时也发现,在写文章时,不管我们如何认真仔细,字处理软件总能出手工无法查出来的错误。

在这个项目中,要求读者开发自己的拼写检查器。我们会提出建议以便帮助读者开始这个项目。读者应该考虑增加更多的功能,同时也会发现用计算机电子词典作为单词源会很有用。

为什么输入时总有很多拼写有错的单词?有时是因为我们记不清单词的正确拼写。有时是因为颠倒了两个字母的顺序(如将“default”敲成“defualt”)。偶尔也会错误地重复输入某个字母(如将“handy”敲成“hanndy”)。有时会敲错了键(如将“birthday”敲成“biryhday”)。这些情况较多,不一一列举。

设计并实现拼写检查器。程序需要使用一个字符串数组 `wordList`。读者要么输入这些字符串,要么从电子词典中获取。

程序首先要求用户输入一个单词,然后在数组 `wordList` 中查找这个单词。找到,则打印“Word is spelled correctly”。

如果找不到输入的单词,程序则打印“Word is not spelled correctly”。然后程序在 wordList 中查找用户可能想输入的单词。例如,可以通过相邻字母的置换发现单词“default”能够匹配 wordList 中的词条。当然,这意味着程序会检查所有其他置换方式的单词如“edfault”,“dfeault”,“deafult”,“defalut”和“defaultl”。找到能匹配 wordList 的新单词时,就打印消息如“Did you mean "default"?”

读者也可实现其他测试如将重复敲入两次的字母替换成单个字母,也可以开发其他功能来改进你的拼写检查器。

# 第 17 章 预处理程序

## 学习目标

- 能够用#include 开发大型程序
- 能够用#define 创建带参数与不带参数的宏
- 理解条件编译
- 能够在条件编译过程中显示错误消息
- 能够用 assert 测试表达式的值是否正确

## 17.1 简介

本章介绍预处理程序。预处理程序发生在程序编译之前,它可能要进行下面的操作:将其他文件包含在要编译的文件中,定义符号常量与宏指令,程序代码的条件编译和预处理程序指令的条件执行。所有的预处理程序指令都以#开头,且一行预处理程序指令前面只能出现空白字符。预处理程序指令不是C++ 语句,因此它们不分号结尾。预处理程序指令在编译之前就已经处理完毕了。

**常见编程错误 17.1** 预处理程序指令用分号结尾可能会导致各种各样的错误,具体情况与预处理程序指令的类型有关。

**软件工程知识 17.1** 许多预处理程序特性(特别是宏)可能更适合 C 程序员,但C++ 程序员也应该熟悉预处理程序因为他们可能需要处理 C 遗留代码。

## 17.2 预处理程序指令#include

预处理程序指令#include 在本书中用得较多,它的作用是将一个指定文件的副本包含到#include指令所处的位置。#include 指令有下面两种形式

```
#include <filename>
#include "filename"
```

两者的区别在于处理器查找要包含的文件的的路径不同。如果文件名放在尖括号里(主要是用于标准库函数的头文件),预处理程序将在预先指定的目录(与系统实现的方式有关)里查找文件。如果文件包含在引号里,预处理程序将首先要在要编译的文件所在目录里查找文件,然后在预先指定的目录(与系统实现方式有关)里查找文件。这种方式通常用于包含程序员自己定义的头文件。

预处理程序指令#include 通常用于包含标准头文件,如<iostream>和<iomanip>,但它也用于将多源文件程序里的源文件包含在一起。我们通常会创建头文件,将分散的程序文



件共用的声明与定义放在一起,并将头文件包含在源文件中。这种声明与定义的例子有类、结构、联合体、枚举和函数原型。

### 17.3 预处理程序指令#define:符号常量

预处理程序指令#define 用于创建符号常量(用符号表示的常量)和宏(用符号定义的操作)。指令格式为

```
#define identifier replacement text
```

文件中出现这行指令时,后续出现的所有该标注符都会在程序编译前自动替换为替换文本。例如

```
#define PI 3.14159
```

会用数值常量 3.14159 替换在这行指令之后出现的所有符号常量 PI。符号常量使程序员能为常量创建名字并将其用于程序。如果需要修改常量,只需修改#define 指令行即可。当程序重新编译时,程序里的这些常量也会随之而改变。注意,符号常量名右边的所有内容都会用于替换符号常量。例如,#define PI =3.14159 会导致预处理程序将程序中位于该指令之后的所有 PI 替换为 =3.14159,这样会导致许多难以察觉的逻辑与语法错误。用新值重定义符号常量也是一个错误。值得注意的是,在C++中,人们更愿意使用常量变量。常量变量有特定的数据类型,调试器可通过其名称访问它。而符号常量一旦被替换,调试器就只能访问替换文本。但常量变量也有一个缺点,它们需要与其数据类型长度相当的内存空间,而符号常量不占用任何额外的内存。

**常见编程错误 17.2** 在定义符号常量的文件之外使用符号常量是语法错误。

**良好编程习惯 17.1** 为符号常量取富于意义的名字有助于提高程序的可读性。

### 17.4 预处理程序指令#define:宏指令

说明:本节内容是为那些需要处理 C 遗留代码的C++程序员设计的。C++中,宏已经被模板与内联函数取代。)宏是预处理程序指令#define 定义的一种操作。与符号常量一样,宏标识符在程序编译前就被替换文本取代。宏可以包含参数,也可以不包含参数。无参数的宏的处理方式与符号常量相同。针对有参数的宏,预处理程序的处理方式是先用替换文本取代参数随后再在程序中展开宏(也就是说,替换文本取代了程序中相应的宏标识符与参数列表)。注意:编译器不会对宏参数进行数据类型检查宏仅用于文本替换。

宏定义

```
#define CIRCLE_AREA(x)(PI*(x)*(x))
```

带有一个计算圆面积参数,无论文件中何处出现 CIRCLE\_AREA(x),宏替换文本中的都会替换为 x 的值。符号常量也替换为前而已定义的值随后在程序中扩展宏。例如,语句

```
area = CIRCLE_AREA(4);
```

在程序中扩展为

```
area = (3.14159*(4)*(4));
```

由于表达式中只包含常量,因此在编译时就计算出了表达式的值,其结果在运行时赋给了变

量 `area`。替换文本 `x` 以及整个表达式两边的圆括号是必需的,它可保证宏参数为表达式时,能以正确的顺序进行计算。例如,语句

```
area = CIRCLE_AREA(c + 2);
```

扩展为

```
area = (3.14159 * (c + 2) * (c + 2));
```

因为圆括号保证了正确的计算顺序,所以计算结果也会正确。如果省略了圆括号,宏的扩展结果便为

```
area = 3.14159 * c + 2 * c + 2;
```

会因为操作优先级规则而错误地求值为如下表达式,从而导致计算不正确。

```
area = (3.14159 * c) + (2 * c) + 2;
```

**常见编程错误 17.3** 未将宏参数包含在括号内是语法错误。

宏 `CIRCLE_AREA` 可以如下定义为函数。函数 `circleArea`

```
double circleArea(double x) {return 3.14159 * x * x;}
```

与宏 `CIRCLE_AREA` 执行相同的操作,但是调用函数会引起系统开销。宏 `CIRCLE_AREA` 的好处是它只是把代码插入程序,避免了函数调用的开销还增强了程序的可续性。同时由于宏 `CIRCLE_AREA` 是分开定义的而且命名为有意义的名字,程序因此也具有较强的可读性。其缺点是宏参数会计算两次。而且,在程序中每次出现宏,就会展开它。如果宏比较大,程序的长度就会因此而变得很长。因此,在程序执行速度与程序长度之间(磁盘空间可能很小)就有一种平衡关系。注意,内联函数(参见第 3 章)由于其宏的性能以及函数在软件工程方面的优点备受程序员青睐。

**性能提示 17.1** 有时宏在程序执行前于代替具有内联代码的函数调用,这样避免了函数调用的开销。但内联函数更可取,因为它们提供了函数的类型检查功能。

宏定义

```
#define RECTANGLE_AREA( x, y ) ( ( x ) * ( y ) )
```

带有两个计算长方形面积参数,不管 `RECTANGLE_AREA( x, y )` 出现在程序的什么地方, `x` 与 `y` 的值都会取代宏替换文本中的 `x` 与 `y`。如此一来,就替换了宏名,对宏进行了扩展。例如,语句

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

扩展为

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

然后计算表达式的值,并将结果赋给变量 `rectArea`。

宏或符号常量的替换文本通常为指令 `#define` 中标识符之后的所有文本。如果它们很长以致无法在同一行显示出来,就可以在行尾使用反斜线(`\`)表明续行。

可用预处理程序指令 `#undef` 取消符号常量与宏的定义。指令 `#undef` 将符号常量或宏进行“撤消定义”。符号常量或宏的作用范围为从其定义处到指令 `#undef` 处或文件尾。一旦撤消定义,就可使用 `#define` 重新定义其名称。

标准函数库中的函数有时也基于其他函数,用宏定义。头文件 `<stdio.h>` 中通常定义宏

```
#define getchar( ) getc( stdin )
```

`getchar` 的宏定义用函数 `getc` 从标准输入流中获得了一个字符。头文件 `<stdio.h>` 中的函数

putchar 以及头文件 `<ctype>` 中的字符处理函数也常用宏来实现。注意带有副作用的表达式(也就是其中的变量值会被修改)不能传递给宏,因为宏参数可能需要计算多次。

## 17.5 条件编译

“条件编译”使程序员能够控制预处理程序指令的执行与程序代码的编译。每个条件预处理程序指令都会计算整型常量表达式以确定程序代码是否应该编译。在预处理程序指令中,不能计算类型转换表达式、sizeof 表达式以及枚举常量。

条件预处理结构与 if 选择结构类似。预处理程序指令

```
#if ! defined ( NULL )
    #define NULL 0
#endif
```

中,这些指令确定常量 NULL 是否已经定义。如果常量 NULL 已经定义,表达式 `! defined ( NULL )` 计算结果为 1,否则为 0。如果结果为 0,则 `! defined ( NULL )` 计算结果就为 1,然后定义 NULL。否则就会忽略 `#define` 指令。每一个 `#if` 结构都以 `#endif` 结束。指令 `#ifdef` 和 `#ifndef` 分别是 `#if defined (名字)` 与 `# ! defined (名字)` 的简写。多重条件预处理结构可用 `#elif` (相当于 if 结构中的 else if) 与 `#else` (相当于 if 结构中的 else) 进行条件测试。

在程序开发中,程序员经常需要注释大块代码以避免编译。如果代码包含 C 风格的注释,那就不能用 `/*` 和 `*/`。但程序员可用预处理结构

```
#if 0
    code prevented from compiling
#endif
```

要想编译代码,只需将上述结构中的 0 替换为 1。

条件编译通常用于辅助调试。我们常用输出语句打印变量值以及确认程序控制流。这些输出语句能放在条件预处理程序指令中以便不在调试程序时编译这些语句。例如

```
#ifdef DEBUG
    cerr << " Variable x = " << x << endl;
#endif
```

在指令 `#ifdef DEBUG` 之前定义符号常量 DEBUG (用 `#define DEBUG`) 时,程序中的 `cerr` 语句才会被编译。调试结束后,就可从源文件中删除 `#define` 指令从而在编译时忽略用于调试目的的输出语句。在稍大的程序中,在源文件不同部分定义不同的符号常量以控制条件编译可能会比较有用。

**常见编程错误 17.4** 在 C++ 只希望出现单条语句的地方插入用于调试目的的语句可能会导致语法错误和逻辑错误。此时,条件编译语句应放在复合语句中。这样一来,程序编译调试语句时,才不会改变程序控制流。

## 17.6 预处理程序指令 #error 与 #pragma

预处理程序指令 `#error`

```
#error tokens
```

用于打印与系统实现有关的消息,其中包括指令中定义的标记。标识是用空格分开的字符序列。例如

```
#error 1 - out of range error
```

包含了 6 个标识。指令 `#error` 在处理过程中,指令中的标识就会以错误消息的形式出现。然后处理终止,并不编译该程序。

预处理程序指令 `#pragma`

```
#pragma tokens
```

也会导致系统实现定义的动作,忽略系统实现不能所识别的。例如,某种 C++ 编译器可能会将程序识别为让程序员能够利用这程序种编译器的特殊功能。关于 `#error` 与 `#pragma` 的更多资料,可参考 C++ 实现文档。

## 17.7 操作符#与##

在 C++ 与标准 C 中都有 `#` 与 `##` 预处理操作符, `#` 操作符将替换文本中的标识转换为引号封闭起来的字符串。程序中出现 `HELLO(John)` 时,宏定义

```
#define HELLO(x) cout << "Hello, " #x << endl;
```

即扩展为

```
cout << "Hello, " "John" << endl;
```

字符串“John”取代了替换文本中的 `#x`。用空格分开的字符串在处理时会连接在一起,因此以上语句等价于

```
cout << "Hello, John" << endl;
```

注意 `#` 操作符必须用于带参数的宏中,因为 `#` 的操作数是宏的参数。

`##` 操作符用于连接两个标识。程序中出现 `TOKENCONCAT` 时,宏定义

```
#define TOKENCONCAT(x,y) x ## y
```

便与其参数即连接在一起取代宏。例如,程序中的 `TOKENCONCAT(O, K)` 会被替换为 `OK`。`##` 操作符必须有两个操作数。

## 17.8 行号

利用 `#line` 预处理程序指令可使随后源代码行从指定整数值开始重新编号。指令

```
#line 100
```

从下一行源代码开始以 100 为基数重新编号。`#line` 指令中可以包含文件名。指令

```
#line 100 "file1.cpp"
```

表明从下一行源代码开始以 100 为基数重新编号,以及任何编译器消息所用的文件名均为“file1.cpp”。这类指令常用于生成更有实际意义的由于语法错误和编译器警告消息。源文件中不会有行号。

## 17.9 预定义符号常量

预定义符号常量有 4 个(如图 17.1 所示)。每个预定义符号常量与开始和结尾均为一

个下划线。这些标识符与已定义的标识符(参见 17.5 节)不能用于指令 `#define` 或 `#undef`。

| 符号常量                  | 说明                                                  |
|-----------------------|-----------------------------------------------------|
| <code>__LINE__</code> | 当前源代码行的行号(一个整数)                                     |
| <code>__FILE__</code> | 假设的源文件名(一个字符串)                                      |
| <code>__DATE__</code> | 源文件的日期编译(一个字符串,格式为“Mmm dd yyyy”的字符串,如“Jan 19 2001”) |
| <code>__TIME__</code> | 源文件的编译时间(一个字符串直接量,格式为“hh:mm:ss”)                    |

图 17.1 预定义的符号常量

## 17.10 宏指令(assert)

头文件 `<cassert>` 中定义的宏 `assert`(断言)用于测试表达式的值。如果表达式的值为 0 (假), `assert` 就会打印一条错误消息并调用通用函数库 `<cstdlib>` 里的函数 `abort` 从而终止程序执行。`assert` 是一种非常有用的工具,用于测试变量是否具有正确值。例如,假定在某程序中变量 `x` 取值绝不会超过 10,此时便可使用 `assert` 来测试变量 `x` 的值并在 `x` 值不正确时打印错误消息。程序遇到语句

```
assert( x <= 10 );
```

时,如果 `x` 的值大于 10,编译器就会打印一条消息,其中包含行号与文件名,然后程序终止。之后程序员可集中精力在相应的区域找到这个错误。如果定义了符号常量 `NDEBUG`,就会省略随后的 `assert`。因此,确定不再需要 `assert` 时(也就是调试结束时),把语句

```
#define NDEBUG
```

插入程序即可而不是手工逐条删除 `assert`。

如今,绝大多数 C++ 编译器都包含异常处理。C++ 程序员更青睐异常处理而不是 `assert`。但是对要处理 C 遗留代码的 C++ 程序员来说, `assert` 仍然具有使用价值。

## 17.11 小结

- 所有预处理程序指令都以 `#` 开始,并在编译程序前处理。
- 预处理程序指令前只能出现空白字符。
- `#include` 指令用于包含指定文件的副本。如果文件名用引号括起来,预处理程序就在编译的文件所在的目录查找要包含的文件。如果文件名用尖括号括起来(`<` 和 `>`),查找就会按照系统实现时定义的方式来进行。
- `#define` 指令用于创建符号常量与宏。
- 符号常量是常量的名字。
- 宏是在 `#define` 预处理程序指令中定义的操作。宏可带参数,也可不带参数。
- 宏或符号常量的替换文本是同一行中指令 `#define` 标识符之后的所有文本。如果它们很长无法在同一行显示,可在行尾用反斜杠(`\`)表示续行。
- 可用“预处理程序指令”`#undef` 撤消符号常量与宏的定义。指令 `#undef` 将符号常量或宏进行“撤消定义”。

- 符号常量或宏的作用范围为从其定义处到指令`#undef`处或文件尾。
- “条件编译”使程序员能控制预处理程序指令的执行与程序代码的编译。
- 条件预处理程序指令计算整型常量表达式确定程序代码是否应该被编译。在预处理程序指令中,不能计算类型转换表达式、`sizeof`表达式以及枚举常量。
- 每个`#if`结构都以`#endif`结束。
- 指令`#ifdef`和`#ifndef`分别是`#if defined(名字)`与`#! defined(名字)`的简写。
- 多重条件预处理结构使用`#elif`与`#else`进行条件测试。
- `#error`指令用于打印与系统实现有关的消息,其中包括指令中定义的标识。
- `#pragma`指令会导致系统中定义的动作。系统不能识别的程序将会被忽略。
- `#`操作符将替换文本中的标识转换为用引号括起来的字符串。`#`操作符必须用于带参数的宏中,因为`#`的操作数为宏的参数。
- `##`操作符用来连接两个标识符`##`。操作符必须有两个操作数。
- `#line`预处理程序指令使得随后的源代码行从指定的整数值开始重新编号。
- 预定义符号常量有 4 个。常量`__LINE__`代表当前源代码行的行号(一个整数)。常量为假设的`__FILE__`源文件名(一个字符串)。常量`__DATE__`为源文件的编译日期(一个字符串)。常量`__TIME__`为源文件的时间编译(一个字符串)。注意它们都以下划线开头和结尾。
- 头文件`<cassert>`中定义的宏`assert`用于测试表达式的值。如果表达式的值为 0 (假),则`assert`会打印一条错误消息并调用通用函数库`<cstdlib>`中的函数`abort`从而终止程序执行。

## 本章术语

argument 参数

\(backslash) continuation character 反斜杠续行符

#concatenation preprocessor operator

连接预处理操作符#

conditional compilation 条件编译

conditional execution of preprocessor

预处理程序的条件执行

convert - to - string preprocessor

转换为字符串预处理程序

debugger 调试器

directives 指令

expand a macro 扩展宏指令

header file 头文件

macro with arguments 带参数的宏指令

macro 宏指令

operator# 操作符#

Predefined symbolic constants 预定义的符号常量

preprocessing directive 预处理程序指令

preprocessor 预处理程序

replacement text 替换文本

scope of a symbolic constant or macro

符号常量或宏指令的作用范围

standard library header files 标准库头文件

symbolic constant 符号常量

## 常见编程错误

17.1 预处理程序指令用分号结尾可能会导致各种各样的错误,具体情况与预处理程序指令的类型有关。

17.2 在定义符号常量的文件之外使用符号常量是语法错误。

17.3 未将宏参数包含在括号内是语法错误。

- 17.4 在C++ 只希望出现单条语句的地方插入用于调试目的的语句可能会导致语法错误和逻辑错误。此时,条件编译语句应放在复合语句中。这样一来,程序编译调试语句时,才不会改变程序控制流。

### 良好编程习惯

- 17.1 为符号常量取富于意义的名字有助于提高程序的可读性。

### 性能提示

- 17.1 有时宏在程序执行前用于代替具有内联代码的函数调用,这样可避免函数调用的开销。但内联函数更可取,因为它们提供了函数类型检查功能。

### 软件工程知识

- 17.1 许多预处理程序特性(特别是宏)可能更适合C 程序员,但C++ 程序员也应该熟悉预处理程序因为他们可能需要处理C 遗留代码。

### 自测题

#### 17.1 填空题:

- a) 每条预处理程序指令都必须以\_\_\_\_\_开始。
  - b) 条件编译结构能用指令\_\_\_\_\_与指令\_\_\_\_\_判断多重情况。
  - c) 指令\_\_\_\_\_用于创建宏与符号常量。
  - d) 预处理程序指令行这前只能出现\_\_\_\_\_字符。
  - e) 指令\_\_\_\_\_用于撤消符号常量与宏名的定义。
  - f) 指令\_\_\_\_\_与指令\_\_\_\_\_为指令`#if defined(名字)`与指令`#if ! defined(名字)`的简写。
  - g) \_\_\_\_\_使程序员能控制预处理程序指令的执行以及程序代码的编译。
  - h) 如果宏\_\_\_\_\_所计算的表达式的值为0,它将打印一条错误消息并终止程序执行。
  - i) 指令\_\_\_\_\_用于在一个文件中插入另外一个文件。
  - j) 操作符\_\_\_\_\_用于连接它的两个参数。
  - k) 操作符\_\_\_\_\_将其参数转换为字符串
  - l) 字符\_\_\_\_\_表明符号常量或宏的替换文本在续行显示。
  - m) 指令\_\_\_\_\_使得随后的源代码行从指定的整数值开始重新编号。
- 17.2 编写程序,打印图 17.1 列出的预定义符号常量的值。
- 17.3 编写预处理程序指令,完成下列操作:
- a) 将符号常量 YES 的值定义为 1。
  - b) 将符号常量 NO 的值定义为 0。
  - c) 包含头文件 `common.h`。头文件与被编译文件处于同一目录。
  - d) 从整数 3 000 开始对文件中剩余的行重新编号。

- e) 如果符号常量 TRUE 已经定义,则撤消其定义,并重定义为 1。不要用预处理程序指令 #ifdef。
- f) 如果符号常量 TRUE 已经定义,则撤消其定义,并重定义为 1。要用预处理程序指令 #ifdef。
- g) 如果符号常量 ACTIVE 不等于 1,则定义符号常量 INACTIVE 为 0。否则将 INACTIVE 定义为 1。
- h) 定义宏 CUBE\_VOLUME 计算立方体的体积(带一个参数)。

### 自测题答案

17.1 a) # b) #elif, #else c) #define d) 空白字符 e) #undef f) #ifdef, #ifndef  
g) 条件编译 h) assert i) #include j) ## k) # l) \ m) #line

17.2 答案如下:

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main()
5 {
6     cout << "__LINE__ = " << __LINE__ << endl;
7     cout << " __FILE__ = " << __FILE__ << endl;
8     cout << " __DATE__ = " << __DATE__ << endl;
9     cout << " __TIME__ = " << __TIME__ << endl;
10     return 0;
11 }

__LINE__ = 6
__FILE__ = C:\ex17_02.cpp
__DATE__ = Apr 28 2000
__TIME__ = 13:48:58
```

17.3 a) #define YES 1  
b) #define NO 0  
c) #include "common.h"  
d) #line 3000  
e) #if defined(TRUE)  
    #undef TRUE  
    #define TRUE 1  
    #endif  
f) #ifdef TRUE  
    #undef TRUE  
    #define TRUE 1  
    #endif  
g) #if ACTIVE  
    #define INACTIVE 0



```

        #else
        #define INACTIVE 1
    #endif
h) #define CUBE_VOLUME( x ) ( ( x ) * ( x ) * ( x ) )

```

## 练习题

17.4 编写程序,定义用于计算球体体积的宏(带一个参数)。程序计算半径从 1 到 10 的球体的体积,并以表格的形式打印结果。球体体积的计算公式为

$$(4.0/3) * \pi * r^3$$

17.5 编写程序,产生输出结果

```
The sum of x and y is 13
```

程序应该定义带两个参数  $x$  与  $y$  的宏 SUM,并用它产生输出。

17.6 编写程序,用 MINIMUM2 确定两个数值中的较小值。从键盘输入这些值。

17.7 编写程序,用 MINIMUM3 确定 3 个数值中的最小值。宏 MINIMUM3 应该用在练习题 17.6 定义的宏 MINIMUM2 确定最小值。通过键盘输入这些值。

17.8 编写程序,用宏 PRINT 打印一个字符串。

17.9 编写程序,用宏 PRINTARRAY 打印整型数组。宏应该将数组和数组中元素的个数作为参数。

17.10 编写程序,用宏 SUMARRAY 计算数值数组中元素值的总和。宏应该将数组和数组中元素的个数作为参数。

17.11 用内联函数重新编写练习题 17.4 ~ 17.10 中的程序。

17.12 如果预处理程序扩展下列宏,可能会产生哪些问题(如果有的话):

- a) #define SQR( x ) x \* x
- b) #define SQR( x ) ( x \* x )
- c) #define SQR( x ) ( x ) \* ( x )
- d) #define SQR( x ) ( ( x ) \* ( x ) )

# 第 18 章 C 遗留代码

## 学习目标

- 能把键盘输入重定向为从文件输入以及将屏幕输出重定向到文件
- 能编写使用变长参数列表的函数
- 能处理命令行参数
- 能处理程序中的意外事件
- 能用类似 C 风格的动态内存分配为数组动态分配内存
- 能用类似 C 风格的动态内存分配动态调整已分配的内存

## 18.1 简介

本章将讲述几个较为高级的主题,入门课程一般不涉及这方面的内容。并且本章讨论的许多功能是某些系统(特别是 UNIX 与 DOS 系统)特有的。了解这里介绍的内容对那些需要处理早期 C 遗留代码的 C++ 程序员有一定的好处。

## 18.2 UNIX 与 DOS 系统中的重定向输入/输出

通常,程序的输入来自于键盘(标准输入),程序的输出是通过屏幕(标准输出)。许多系统(特别是 UNIX 与 DOS 系统)中,可以将输入重定向为从文件输入,将输出重定向到文件。这两种重定向都不需要使用标准函数库中的文件处理函数。

UNIX 命令行的重定向输入/输出有几种方式。以可执行文件 `sum` 为例,它要输入几个整数,一次输入一个,并每次都要计算出它们的和直到碰到文件终止符,然后打印结果。通常用户从键盘输入这些整数,然后可通过文件终止组合键表明输入结束。通过输入重定向,要输入的数据可以存储在文件里。例如,假设数据存储在文件 `input` 里,命令行

```
$ sum < input
```

就会开始执行文件 `sum`。“重定向输入标识符”( `<` )表明文件 `input` (取代了键盘)中的数据充当了程序输入。DOS 系统中的重定向输入也与此类似。

注意, `$` 为 UNIX 的命令行提示符(有些 UNIX 系统将 `%` 用作提示符)。学生们经常发现很难把重定向理解为操作系统函数而不是 C++ 的特征。

输入重定向的另一种方法是建立管道。管道(`|`)可使一个程序的输出重定向为另一个程序的输入。假定程序 `random` 输出一系列随机整数,那就可用 UNIX 命令行

```
$ random | sum
```

将 `random` 的输出通过“管道”直接重定向到程序 `sum`。这样一来就能计算出 `random` 产生的各整数之和。UNIX 与 DOS 系统中都可以建立管道。

可用“重定向输出标识符”( > ,UNIX 与 DOS 系统使用的符号相同)将程序输出重定向到文件。例如,要想将程序 random 的输出重定向到文件 out,可用命令行

```
$ random > out
```

最后,还可用“附加输出符”( >> ,UNIX 与 DOS 系统使用的符号相同)将程序输出附加到现有文件结尾处。例如,将程序 random 的输出附加到前一个命令行创建的文件 out 结尾处,可使用命令行

```
$ random >> out
```

### 18.3 变长参数列表

注意:这里介绍的材料对要处理 C 遗留代码的 C++ 程序员来说,非常有用。针对 C 程序须用变长参数列表来完成的大多数工作, C++ 程序员可用函数重载来完成。可创建函数接收未指定个数的参数。函数原型中的省略号(...)表明函数接收个数可变的任意类型的参数。注意,省略号必须位于参数列表的最后,而且至少有一个已命名的参数。变长参数头文件 <stdarg.h> (如图 18.1 所示)中的宏与定义提供了构造带有变长参数列表的函数所需的功能。

| 标识符      | 说明                                                                      |
|----------|-------------------------------------------------------------------------|
| va_list  | 用于保存宏 va_start, va_arg 与 va_end 所需信息的类型。要想访问变长参数列表,必须声明一个 va_list 类型的对象 |
| va_start | 访问在变长参数列表前调用的宏。它用于初始化声明为 va_list 类型的对象供宏 va_arg 与宏 va_end 使用            |
| va_arg   | 扩展为表达式的宏,其中包含变长参数列表中下一参数的值与类型。每次调用它都会修改声明为 va_list 类型的对象以使对象指向下一个参数     |
| va_end   | 该宏使用了一个函数(前题是 va_start 宏引用了该函数的变长参数列表)所返回的普通值                           |

图 18.1 在头文件 <stdarg.h> 中定义的类型与宏

图 18.2 演示了该函数将函数 average 的用法接收个数可变的参数。它的第一个参数始终是参数个数的平均数。

```

1 //Fig.18.2: fig18_02.cpp
2 //Using variable-length argument lists
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8
9 #include <iomanip>
10
11 using std::setw;
12 using std::setprecision;
13 using std::setiosflags;
14
15 #include <stdarg.h>
```

```

16
17 double average( int, ... );
18
19 int main()
20 |
21     double w = 37.5, x = 22.5, y = 1.7, z = 10.2;
22
23     cout << setiosflags( ios::fixed | ios::showpoint )
24         << setprecision( 1 ) << "w = " << w << " \nx = " << x
25         << " \ny = " << y << " \nz = " << z << endl;
26     cout << setprecision( 3 ) << " \nThe average of w and x is "
27         << average( 2, w, x )
28         << " \nThe average of w, x, and y is "
29         << average( 3, w, x, y )
30         << " \nThe average of w, x, y, and z is "
31         << average( 4, w, x, y, z ) << endl;
32     return 0;
33 }
34
35 double average( int i, ... )
36 |
37     double total = 0;
38     va_list ap;
39
40     va_start( ap, i );
41
42     for ( int j = 1; j <= i; j++ )
43         total += va_arg( ap, double );
44
45     va_end( ap );
46
47     return total / i;
48 |

```

输出结果:

```

w=37.5
x=22.5
y= 1.7
z=10.2

```

```

The average of w and x is 30.000
The average of w,x,and y is 20.567
The average of w,x,y,and z is 17.975

```

图 18.2 变长参数列表用法示例

函数 `average` 使用了头文件 `<cstdarg>` 中所有的定义与宏。宏 `va_start`, `va_arg` 和 `va_end` 用 `va_list` 类型的对象 `ap` 处理函数 `average` 的变长参数列表。函数调用 `va_start` 初始化对象 `ap` 以供宏 `va_arg` 与 `va_end` 使用。宏 `va_start` 接收两个参数(对象 `ap` 与省略号前最右边的那个参数),这里为 `i`(`va_start` 用 `i` 来确定变长参数的开始位置)。接着,函数 `average` 将变长参数列表中的参数逐个加入变量 `total` 中。通过调用宏 `va_arg`,可从参数列表

中获得要相加的值。宏 `va_arg` 接收两个参数,对象 `ap` 与参数列表中当前值的类型(这里为 `double`),并返回这个参数的值。函数 `average` 调用其参数为对象 `ap` 的宏 `va_end`,使函数 `average` 正常返回主函数 `main`。最终,函数会计算平均值将其返回函数 `main`。注意,在这里我们在参数列表的可变长度部分中只用了 `double` 类型的参数。实际上,只要每次调用 `va_arg` 时指定正确的类型,就可以使用任何一种数据类型或几种类型的混合。

**常见编程错误 18.1** 将省略号放在函数参数列表中,省略号只能放在参数列表最后。

## 18.4 使用命令行参数

在许多系统中(特别是 UNIX 与 DOS)。将参数 `int argc` 与 `char * argv[ ]` 包含在函数 `main` 的参数列表中,令命令行将参数传递给函数 `main`。参数 `argc` 接收命令行参数的个数。参数 `argv` 是一个字符串数组,其中存储了实际的命令行参数。命令行参数的常见用法包括打印参数,将选项传递给程序以及将文件名传递给程序。

图 18.3 中的程序以一次复制一个字符的方式,将一个文件复制到另一个文件中。该程序的可执行文件名为 `copy`。UNIX 系统中,程序 `copy` 所用的典型命令行是

```
$ copy input output
```

它表示把文件 `input` 复制到文件 `output` 中。程序执行时,如果其参数不为 3(`copy` 作为其中一个参数),程序便打印一条错误消息然后终止。否则,数组 `argv` 就会包含字符串“`copy`”、“`input`”与“`output`”。命令行中的第二个与第 3 个参数为程序使用的文件名。这两个文件的打开可通过创建 `ifstream` 对象 `inFile` 与 `ofstream` 对象 `outFile` 来实现。如果它们都能成功打开,就可用成员函数 `get` 从文件 `input` 读取字符并用成员函数 `put` 将这些字符写入文件 `output`,直到碰到文件 `input` 的文件结束标志为止。然后程序终止。结果为与文件 `input` 的副本。注意,并非所有计算机系统都像 UNIX 与 DOS 那样很轻松地支持命令行参数。有些系统,如 Macintosh 与 VMS,需要专门的配置来处理命令行参数。有关命令行参数的详情,请参考系统手册。

```
1 //Fig.18.3: fig18_03.cpp
2 //Using command-line arguments
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::ios;
8
9 #include <fstream>
10
11 using std::ifstream;
12 using std::ofstream;
13
14 int main( int argc, char *argv[] )
15 {
16     if ( argc != 3 )
17         cout << "Usage: copy infile outfile" << endl;
```

```

18     else {
19         ifstream inFile( argv[ 1 ], ios::in );
20
21         if ( ! inFile ) {
22             cout << argv[ 1 ] << " could not be opened" << endl;
23             return -1;
24         }
25
26         ofstream outFile( argv[ 2 ], ios::out );
27
28         if ( ! outFile ) {
29             cout << argv[ 2 ] << " could not be opened" << endl;
30             inFile.close();
31             return -2;
32         }
33
34         while ( ! inFile.eof() )
35             outFile.put( static_cast< char >( inFile.get() ) );
36
37         inFile.close();
38         outFile.close();
39     }
40
41     return 0;
42 }

```

图 18.3 命令行参数用法示例

## 18.5 编译多个源文件程序的相关说明

本书前面曾讲过,可以创建包含多个源文件的程序(参见第6章)。创建多文件程序时,有一些地方需要注意,如函数的定义必须完全包含在一个文件中——而不能跨两个或两个以上的文件。

在第3章,我们介绍了存储类别与作用范围的概念。我们知道,在任何函数定义以外声明的变量具有默认的静态存储类,而且全局变量可以被同一个文件中在其声明之后的任何函数访问。其他文件中的函数也可以访问全局变量。但每个打算使用全局变量的文件都要事先声明该全局变量。例如,如果我们在一个文件中定义了全局整型变量 `flag`,并打算在要在另一个文件中引用它,第二个文件在使用该全局变量之前必须在包含声明

```
extern int flag;
```

以上声明中,存储类别说明符 `extern` 告诉编译器变量 `flag` 要么在同一文件的后面部分定义,要么在另一个文件中定义。然后编译器就告诉连接程序,文件中出现了对变量 `flag` 的未能解决的引用(编译器不知道变量 `flag` 定义于何处,所以让连接程序查找变量 `flag`)。如果连接程序找不到变量 `flag`,它就会报告一条错误消息同时不会产生任何可执行文件。如果找到正确的全局变量声明,就指向变量 `flag` 的位置确定那些未确定的引用。

**性能提示 18.1** 全局变量能够提高性能,因为它们能够被所有函数直接访问,消除了将数

据传递给函数的开销。

**软件工程知识 18.1** 尽量避免使用全局变量,除非应用程序时性能是关键因素时,因为全局变量违背了最低权限原则,加大了维护软件的难度。

正如 `extern` 声明能够用于在其他文件中声明全局变量一样,函数原型也能将一个函数的作用范围扩展到定义它的文件之外(在函数原型中,无需使用 `extern`)。具体作法是:把将函数原型包含在调用它的文件中,然后将它们编译在一起(参见 17.2 节)。函数原型告诉编译器指定函数要么在同一文件的后面部分定义,要么在另一个文件中定义。编译器不会去确定对这些函数的引用,而是把任务留给连接程序。如果连接程序找不到函数定义,就会产生错误。

作为用函数原型扩展函数作用范围的例子,我们可以考虑任一个其中包含预处理程序指令 `#include <cstring>` 的程序。预处理程序指令将函数(如 `strcmp` 与 `strcat`)的函数原型包含在一个文件中。这个文件中的其他函数就可用函数 `strcmp` 与 `strcat` 来执行任务。对我们来说,函数 `strcmp` 与 `strcat` 是单独定义的。我们不需要知道它们在何处定义,我们只是在程序中重用代码而已。连接程序会确定对这些函数的引用。这种处理使我们可以利用标准函数库中的函数。

**软件工程知识 18.2** 创建多个源文件的程序有助于软件重用和良好的软件工程实施。函数可能供多个应用程序共同使用。在这种情况下,这些函数应该存放在它们自己的源文件中。每个源文件都有其相应的包含函数原型的头文件。这样一来,不同应用程序的程序员就能够重复使用同样的源代码,只要他们在程序中包含恰当的头文件,并把自己的应用程序和相应的源文件一起编译。

**可移植性提示 18.1** 有些系统不支持长度超过 6 个字符的全局变量名或函数名。编写应用到多平台的程序时应该考虑到这一点。

可以将全局变量或函数的作用范围限定在定义它们的文件里。对一个全局变量或一个函数使用存储类别说明符 `static` 时,即可防止它们被其他文件中定义的函数使用。这称为“内部连接”。没有使用 `static` 的全局变量或函数则为“外部连接”——如果其他文件包含了正确的声明或函数原型,就能够在这些文件访问到这些全局变量和函数。

#### 全局变量声明

```
static double pi = 3.14159;
```

创建了 `double` 类型的变量 `pi`,并将其初始化为 3.14159,同时也表明 `pi` 只能够被定义它的同一个文件中的函数访问。

`static` 说明符通常与工具函数一起使用,这些工具函数只能被特定文件中的函数调用,用于只被特定文件里的函数调用的功能函数中。如果函数不需要用于特定文件以外,就应该使用 `static` 对它实施最低权限原则。如果文件中一个函数在使用之前定义,就应该在函数定义中使用 `static`,否则就应该在函数原型中使用 `static`。

构建多个源文件的大型程序时,会因为一个小小的改动而导致编译过程变得冗长而乏味。许多系统提供了实用程序,用于只重新编译有改动的程序文件。UNIX 系统中,这种实用程序称为 `make`。它读取其中包含编译与连接程序所用指令的,称为 `makefile` 的文件。有

些系统如用于 PC 机的 Borland C++ 与 Microsoft Visual C++ 都提供了 make 程序和“项目”。关于 make 程序的更多信息,请参考系统使用手册。

## 18.6 用函数 exit 与 atexit 终止程序运行

通用函数库(cstdlib)提供了终止程序执行的方法。此类方法不同于函数 main 正常返回。函数 exit 强迫程序终止就好像程序正常执行一样。它常用于检测到输入错误时或者不能打开程序要处理的文件时终止程序。函数 atexit 用于在程序中“注册”一个函数,程序将在成功终止之前调用这个函数——也就是说,程序将在抵达 main 函数结尾或调用 exit 时终止执行。

函数 atexit 将指向一个函数的指针(也就是函数名)作为其参数。程序终止时调用的函数不能带参数且不能返回值。在程序终止时最多能注册 32 个函数。

函数 exit 只有一个参数。参数一般为符号常量 EXIT\_SUCCESS 或 EXIT\_FAILURE。如果 exit 随参数 EXIT\_SUCCESS 一起调用,那就会把成功终止执行的实现值返回调用环境。如果 exit 随参数 EXIT\_FAILURE 一起调用,就会返回执行不成功的实现值。调用函数 exit 时,以前用 atexit 注册的所有函数都会按其注册顺序相反的顺序被调用。与程序有关的所有流都会被清空并关闭。然后控制返回到主环境。图 18.4 测试了函数 exit 与 atexit。程序提醒用户确定用 exit 终止程序还是到达函数 main 结时终止程序。注意,在这两个情况下,程序终止时都会执行函数 print。

```

1 //Fig.18.4: fig18_04.cpp
2 //Using the exit and atexit functions
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::cin;
8
9 #include <cstdlib>
10
11 void print( void );
12
13 int main()
14 {
15     atexit( print ); //register function print
16     cout << "Enter 1 to terminate program with function exit "
17          << " \nEnter 2 to terminate program normally \n";
18
19     int answer;
20     cin >> answer;
21
22     if ( answer == 1 ) {
23         cout << " \nTerminating program with function exit \n";
24         exit( EXIT_SUCCESS );
25     }

```



```

26
27     cout << "\nTerminating program by reaching the end of main"
28         << endl;
29
30     return 0;
31 }
32
33 void print( void )
34 {
35     cout << "Executing function print at program termination\n"
36     << "Program terminated" << endl;
37 }

```

输出结果:

```

Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1

Terminating program with function exit
Executing function print at program termination
Program terminated

Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated

```

图 18.4 函数 exit 与 atexit 用法示例

## 18.7 类型限定符 volatile

类型限定符 volatile 用于定义可在程序外部改变的变量(也就是说,这种变量不完全由程序控制)。因此编译器不能不能对其优化(例如,加快程序执行速度或减小内存消耗),因为优化需要知道“变量行为只受编译器能观察得到的程序活动影响”。

## 18.8 整数和浮点数常量的后缀

C++ 提供了整数和浮点数后缀来指定整数和浮点数常量的类型。整数后缀有:u 或 U 表示 unsigned 整数,l 或 L 表示 long 整数,ul 或 UL 表示 unsigned long 整数。常量类型

```

174u
8385L
28373ul

```

分别为 unsigned 型、long 型和 unsigned long 型:如果整数常量没有后缀,那么其类型便能够存储它的第一种类型(首先是 int,然后是 long int,最后是 unsigned long int)。

浮点数的后缀为:f 或 F 表示 float 型值,l 或 L 表示 long double 型值。常量类型

3.14159L

1.28f

分别为 long double 和 float, 无后缀的浮点常量类型为 double。后缀不正确的常量会引起编译器警告或错误消息。

## 18.9 信号处理

意外事件或信号能提前终止程序的执行。意外事件包括中断(在 UNIX 或 DOS 系统中意外键入了按 Ctrl-c)、非法指令、段非法访问、来自操作系统的终止指令和浮点异常(被 0 除或与很大的浮点数相乘)。“信号处理函数库”提供了函数 signal 来捕捉意外事件。函数 signal 接收两个参数:一个整型信号数和一个指向信号处理的指针。信号可由函数 raise 产生, 函数 raise 将整型信号数作为参数。图 18.5 小结了头文件 <csignal> 中定义的标准信号。图 18.6 中的程序演示了函数 signal 与 raise 的用法。

| 信号      | 说明                       |
|---------|--------------------------|
| SIGABRT | 程序异常终止(如调用函数 abort)      |
| SIGFPE  | 错误的算术运算,如被 0 除或导致结果溢出的操作 |
| SIGILL  | 检测到非法指令                  |
| SIGINT  | 收到交互式的提醒信号               |
| SIGSEGV | 非法访问内存                   |
| SIGTERM | 发送给程序的终止请求               |

图 18.5 头文件 <csignal> 中定义的信号

图 18.6 中的程序用函数 signal 来捕捉一个交互信号(SIGINT)。程序用 SIGINT 和指向函数 signal\_handler 的指针(记住,函数名即指向函数的指针)作为参数调用函数 signal。因此,有 SIGINT 类型的信号产生时,程序就会调用函数 signal\_handler,并打印一条消息,同时询问用户是否继续程序的正常执行。如果用户想继续执行,信号处理器就会再次调用 signal 重新进行初始化(有些系统要求信号处理器重新初始化),然后控制返回检测到信号的位置。在本程序中,函数 raise 用于模拟产生一个交互信号。程序在 1 到 50 之间随机选择一个整数。如果这个数为 25,程序就调用函数 raise 产生交互信号。一般情况下,交互信号在程序外部进行初始化。例如在 UNIX 或 DOS 系统中,如程序执行时按 Ctrl-c 就会产生一个终止程序执行的交互信号。信号处理能用来捕捉这种信号并防止程序终止。

```

1 //Fig.18.6: fig18_06.cpp
2 //Using signal handling
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
```

```
12
13 #include <csignal>
14 #include <cstdlib>
15 #include <ctime>
16
17 void signal_handler( int );
18
19 int main()
20 {
21     signal( SIGINT, signal_handler );
22     srand( time( 0 ) );
23
24     for( int i = 1; i < 101; i++ ) {
25         int x = 1 + rand() % 50;
26
27         if ( x == 25 )
28             raise( SIGINT );
29
30         cout << setw( 4 ) << i;
31
32         if ( i % 10 == 0 )
33             cout << endl;
34     }
35
36     return 0;
37 }
38
39 void signal_handler( int signalValue )
40 {
41     cout << "\nInterrupt signal ( " << signalValue
42         << " ) received.\n"
43         << "Do you wish to continue (1 = yes or 2 = no)? ";
44
45     int response;
46     cin >> response;
47
48     while ( response != 1 && response != 2 ) {
49         cout << "(1 = yes or 2 = no)? ";
50         cin >> response;
51     }
52
53     if ( response == 1 )
54         signal( SIGINT, signal_handler );
55     else
56         exit( EXIT_SUCCESS );
57 }
```

输出结果:

```
1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

```

31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88
Interrupt signal (4) received.
Do you wish to continue (1 = yes or 2 = no)? 1
89 90
91 92 93 94 95 96 97 98 99 100

```

图 18.6 信号处理用法示例

## 18.10 用 calloc 与 realloc 动态内存分配

在第 7 章,我们讨论了用 new 与 delete 实现的 C++ 风格动态内存分配,并比较了 new 与 delete 和 C 风格的函数 malloc 与 free。C++ 程序员应使用 new 和 delete,而不是 malloc 和 free。但大多数 C++ 程序员都需要面对大量的 C 遗留代码。因此,我们将讨论 C 风格的动态内存分配。

通用函数库 <csdlib> 提供了另外两个用于动态内存分配的函数: calloc 和 realloc。它们可用于创建和修改“动态数组”。第 5 章曾提到,数组的指针能够象数组那样带下标。因此,可以像操作数组那样,操作指向 calloc 创建的连续存储内存区的指针。函数 calloc 为数组动态分配内存并自动将它们初始化为 0。其函数原型为

```
void *calloc( size_t nmemb, size_t size );
```

它接收两个参数:元素的个数(nmemb)与各元素的长度(size),并将数组中的元素初始化为 0。函数返回指向已分配内存的指针,但如果内存没有分配,则返回空指针(0)。

函数 realloc 修改前一次调用 malloc、calloc 或 realloc 所分配的对象长度。如果分配的内存比以前分配的内存大,原始对象的内容就不会改变。否则,原始对象中的内容就不会改为新对象的长度。其函数原型为

```
void *realloc( void *ptr, size_t size );
```

函数 realloc 带两个参数,指向原来对象的指针(ptr)和重新分配给对象的空间的长度(size)。如果 ptr 为 0, realloc 与函数 malloc 的作用就相同。如果 size 为 0 且 ptr 不为 0,对象的内存就会被释放。或者 ptr 不为 0 且 size 大于 0, realloc 会试图分配新的内存块。如果无法分配新内存, ptr 指向的对象将保持不变。函数 realloc 返回指向重分配内存的指针或空指针。

## 18.11 无条件转向语句: goto

我们一直都在强调利用结构编程技巧构建易于调试、维护和修改的软件的重要性。在有些情况下,性能比严格遵守结构化编程技巧更为重要。这时就可使用非结构化的编程技巧。例如,可以使用 break 在循环继续条件为假之前终止循环执行。如果任务在循环终止之前完成,便可节省不必要的循环。

另一个非结构化编程的例子是 goto 语句——无条件转向语句。goto 语句将程序控制流

转到 goto 语句指定的标号后的第一条语句。标签号是后面跟有一个冒号的标识符。标号必须与引用它的 goto 语句处理同一个函数中。图 18.7 中的程序用 goto 语句循环 10 次并打印每次循环的计数值。程序将 count 初始化为 1, 然后测试 count 是否大于 10 (标号被忽略, 因为它不执行任何操作)。如果 count 大于 10, 程序则从 goto 语句处转到标号 end 后的第一条语句。否则, 打印 count 值并进行递增, 然后程序控制便从 goto 语句处转到标号 start 后的第一条语句。

```

1 //Fig.18.7: fig18_07.cpp
2 //Using goto
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int count = 1;
11
12     start;                //label
13     if ( count > 10 )
14         goto end;
15
16     cout << count << " ";
17     ++count;
18     goto start;
19
20     end;                  //label
21     cout << endl;
22
23     return 0;
24 }
```

输出结果:

```
1 2 3 4 5 6 7 8 9 10
```

图 18.7 goto 语句用法示例

第 2 章, 我们提到仅用 3 种控制结构 (顺序、选择与循环) 就可以编写所有程序。遵守结构化编程规则时, 我们有可能会创建嵌套级别较深且很难高效率地跳出的控制结构。这时, 有些程序员就会用 goto 语句跳出这种结构。如此一来, 在测试多重条件时, 就没有必要跳出控制结构。

**性能提示 18.2** goto 语句能用于高效率跳出嵌套级别很深的控制结构。

**软件工程知识 18.3** goto 语句只适用于追求性能的应用程序。它是非结构化的, 可能会导致程序难于调试、维护和修改。

## 18.12 联合体

用关键字 union 定义的联合体是一块内存区域, 它在不同的时间里可能包含不同类型的

对象。但任何时间,联合体最多只能包含一个对象,因为联合体的成员共享同一块存储空间。因此,确保用数据类型正确的成员名来引用联合体便是程序员的职责。

**常见编程错误 18.2** 引用非最后一个存储的联合体成员时,会产生不确定的结果。这会将存储数据视为另一种类型。

**可移植性提示 18.2** 如果数据在联合体中,以一种类型进行存储,而以另一种类型被引用,会产生与系统实现过程有关的结果。

在程序执行的不同时间里,虽然有些对象是相关的,但也有些对象不相关。因此可用联合体来共享存储空间,避免了为不需要对象浪费空间。存储联合体的字节数应该足以容纳所占空间最大的成员。

**性能提示 18.3** 使用联合体可节省存储空间。

**可移植性提示 18.3** 存储一个联合体所需的空间与系统实现有关。

**可移植性提示 18.4** 有些联合体可能不能很好地移植到其他计算机系统中。联合体是否可以移植与稳定系统中,联合体成员的数据类型的存储对齐要求有关。

声明联合体的格式与结构和类相同。例如

```
union Number {  
    int x;  
    double y;  
};
```

表示 Number 为拥有成员 int x 与 double y 的联合体。程序中,联合体的定义通常位于函数 main 之前,因此程序中所有函数都可用它来声明变量。

**软件工程知识 18.4** 如同结构或类声明,联合体声明只是创建了一种新的类型。将联合体或结构声明放在任何函数之外并不会创建全局变量。

能够用于联合体的有效内部操作只包括:将一个联合体赋给同类型的另一个联合体,取联合体的地址(&)以及用结构成员操作符(.)与结构指针操作符(->)访问联合体成员。与结构不能进行比较的原因相同,联合体也不能进行比较。

**常见编程错误 18.3** 比较联合体会导致语法错误,因为编译器不知道联合体的当前成员为哪一个,从而不知道应该比较哪两个成员。

联合体可用构造函数来初始化其成员,这一点与类相似。没有构造函数的联合体可以用同类型的另一个联合体来初始化,可用与另一个联合体的第一个成员类型相同的表达式或初始化器(包含在括号中)来初始化。联合体也可以有其他成员函数如析构函数。但联合体的成员函数不能声明为虚拟函数。联合体成员默认为 public。

**常见编程错误 18.4** 在联合体声明中,用其类型不同于联合体第一个成员的值或表达式来初始化联合体。

联合体不能用作继承中的基类(即类不能从联合体派生而来)。联合体可以有对象成

员,但这类对象不能有构造函数、析构函数或重载赋值操作符。联合体的数据成员不能声明为 static。

图 18.8 中的程序用联合体 number 类型的变量 value 展示联合体中可保存整型和双精度型的数值。程序输出结果与系统实现有关。程序输出表明双精度值与整型值的内部表示差别较大。

```

1 //Fig.18.8: fig18_08.cpp
2 //An example of a union
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 union Number {
9     int x;
10    double y;
11 };
12
13 int main()
14 {
15     Number value;
16
17     value.x = 100;
18     cout << "Put a value in the integer member \n"
19          << "and print both members. \nint:  "
20          << value.x << " \ndouble: " << value.y << " \n\n";
21
22     value.y = 100.0;
23     cout << "Put a value in the floating member \n"
24          << "and print both members. \nint:  "
25          << value.x << " \ndouble: " << value.y << endl;
26     return 0;
27 }
```

输出结果:

```

Put a value in the integer member
and print both members.
int: 100
double: -9.25596e+061

Put a value in the floating member
and print both members.
int: 0
double: 100
```

图 18.8 用两种成员数据类型打印一个联合体的值

“匿名”联合体没有类型名,同时在其表示终止的分号前也没有定义对象或指针。这种联合体没有创建类型,而是创建了一个未命名的对象。“匿名”联合体能够在其声明的范围内直接访问,和访问局部变量一样(不需要用点(.)或箭头(->)操作符)。

“匿名”联合体有一定的限制。它只能包含数据成员且其所有成员都是 public 成员。

声明为全局作用范围(即文件作用范围)的“匿名”联合体必须显式声明为 `static`。图 18.9 演示了“匿名”联合体的用法。

```

1 //Fig.18.9: fig18_09.cpp
2 //Using an anonymous union
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 |
10     //Declare an anonymous union.
11     //Note that members b, d, and fPtr share the same space.
12     union {
13         int b;
14         double d;
15         char *fPtr;
16     };
17
18     //Declare conventional local variables
19     int a = 1;
20     double c = 3.3;
21     char *ePtr = "Anonymous";
22
23     //Assign a value to each union member
24     //successively and print each.
25     cout << a << " ";
26     b = 2;
27     cout << b << endl;
28
29     cout << c << " ";
30     d = 4.4;
31     cout << d << endl;
32
33     cout << ePtr << " ";
34     fPtr = "union";
35     cout << fPtr << endl;
36
37     return 0;
38 |

```

输出结果:

```

1 2
3.3 4.4
Anonymous union

```

图 18.9 “匿名”联合体用法示例

## 18.13 接合规约

C++ 程序中可在调用 C 编译器中编写和编译的函数。3.20 节中提到, C++ 为实现类型



安全连接专门对函数名进行了编码。然而 C 并不对它的函数名进行编码。因此,试图连接 C 代码与 C++ 代码时,C 中编译的函数就会无法识别,因为 C++ 代码需要的是经过专门编码的函数名。C++ 能够通过程序员提供的“接合规约”(linkage specifications)通知编译器函数是在 C 编译器中编译的,以防止了 C++ 编译器对函数名进行编码。开发专用的大型函数库时,接合规约非常有用。这些情况下,用户不会为了在 C++ 中重新编译源代码而访问它们,或者不必花时间将这些库函数移植到 C++。

要想通知编译器一个或多个函数是在 C 中编译的,可以写以下函数原型

```
extern "C" function prototype //single function
extern "C" //multiple functions
{
    function prototypes
}
```

这些声明语句声明通知编译器规定函数不是在 C++ 中编译的,不需要对这些函数名进行编码。然后这些函数能够正确地与程序接合。C++ 环境中通常包含了标准 C 函数库,因而不需要程序员对这些函数使用接合规约。

## 18.14 小结

- 在许多系统(特别是 UNIX 与 DOS 系统)中,可以将输入重定向为文件输入,将输出重定向到文件。UNIX 与 DOS 命令行中可使用重定向输入符(<)或使用管道(|)进行重定向。UNIX 与 DOS 命令行中可使用重定向输出符(>)或使用附加输出符(>>)进行重定向。重定向输出符将程序输出存储到文件中而附加输出符将输出附加到文件名尾。
- 变长参数头文件 <stdarg.h> 中的宏与定义提供了构造带有变长参数列表的函数所必需的功能。
- 在函数原型中的省略号(...)表明函数接收任意类型个数可变的参数。
- 类型 va\_list 用于保存宏 va\_start, va\_arg 与 va\_end 所需信息。要想访问变长参数列表,必须声明一个 va\_list 类型的对象。
- 可以在访问变长参数列表之前调用宏 va\_start。它用于初始化声明为 va\_list 类型的对象以供宏 va\_arg 与宏 va\_end 使用。
- 宏 va\_arg 扩展为包含了可变量参数列表中下一个参数的值与类型的表达式。每次调用它都会修改声明为类型 va\_list 的对象以使对象指向下一个参数。
- 宏 va\_end 使用函数正常返回的值,这类函数的变长参数列表被 va\_start 引用。
- 在许多系统中(特别是 UNIX 与 DOS)可将参数 int argc 与 char \*argv[] 包含在函数 main 的参数列表中,从而从命令行将参数传递给函数 main。参数 argc 接收命令行参数的个数。参数 argv 为字符串数组,其中存储了实际的命令行参数。
- 函数的定义必须完全包含在文件中,而不能跨越两个或两个以上的文件。
- 全局变量必须在每一个使用它的文件里进行声明。
- 函数原型也能够将函数的作用范围扩展到定义它的文件之外(在函数原型中,无需

使用 `extern` 说明符)。具体做法是:将函数原型包含在调用它的文件中,然后将其编译在一起。

- 对一个全局变量或一个函数使用存储类别限定符 `static` 时,它可以防止它们被另一个文件中定义的函数使用。这称之为“内部连接”。没有使用 `static` 进行限定的全局变量或函数则为“外部连接”。如果文件包含了正确的声明或函数原型,这些文件就访问。
- `static` 说明符通常为工具函数,只能供特定文件里的函数调用。如果在一个特定文件之外不需要使用函数,就应该使用 `static` 对它实施最低权限原则。
- 构建多个源文件的大型程序时,如果一个小小的改动而重新编译整个程序,编译过程将变得冗长而乏味。许多系统提供了实用程序只重新编译有改动的程序文件。UNIX 系统中,这种实用程序被称为 `make`。它读取包含了编译与连接程序所用指令的,称为 `makefile` 的文件。
- 函数 `exit` 强迫程序在正常情况下终止执行。
- 函数 `atexit` “注册”程序成功终止时调用的函数,也就是说,在程序到达函数 `main` 末尾或调用 `exit` 时。
- 函数 `atexit` 将指向一个函数的指针(也就是函数名)作为其参数。程序终止时能够调用的函数不能带参数且不能返回值。在程序终止时最多只能注册 32 个函数。
- 函数 `exit` 只有一个参数。参数一般为符号常量 `EXIT_SUCCESS` 或 `EXIT_FAILURE`。如果 `exit` 随参数 `EXIT_SUCCESS` 一起调用,它会返回成功终止时的实现值并将它返回调用环境。如果 `exit` 随参数 `EXIT_FAILURE` 一起调用,就返回未成功终止时的实现值。
- 调用函数 `exit` 时,之前用函数 `atexit` “注册”的函数就以与“注册”顺序相反的顺序调用。与程序有关的所有流都会被清空并关闭。然后控制便返回主环境。
- 限定符 `volatile` 用于防止变量的优化,因为变量可以在程序范围之外被修改。
- C++ 提供了整数和浮点数后缀指定整数和浮点数常量的类型。整数后缀有: `u` 或 `U` 表示无符号整数, `l` 或 `L` 表示长整型整数, `ul` 或 `UL` 表示无符号长整型整数。如果一个整型常量没有后缀,那么其类型由能够存储它的第一种类型决定(首先是 `int`, 然后是 `long int`, 最后是无符号 `long int`)。浮点型后缀为: `f` 或 `F` 表示 `float` 型数, `l` 或 `L` 表示 `long double` 型数。无后缀的浮点常量类型为 `double`。
- 信号处理函数库提供了函数 `signal` 来捕捉无法预料的事件。函数 `signal` 接收两个参数:一个整型信号数和一个指向信号处理的指针。
- 通用函数库(`cstdlib`)提供了两个用于动态内存分配的函数: `calloc` 和 `realloc`。它们能用于创建动态数组。
- 信号能够通过函数 `raise` 和一个整型参数产生。
- 函数 `calloc` 接收两个参数:元素的个数(`nmemb`)与每个元素的长度(`size`),并将数组中的元素初始化为 0。函数返回指向已分配数组的指针,但如果没有分配内存,则返回空指针(0)。
- 函数 `realloc` 改变之前调用 `malloc`, `calloc` 或 `realloc` 所分配的对象的大小。如果分配

的内存比先前分配的内存要大,原始对象的内容不会改变。

- 函数 `realloc` 带两个参数:指向原始对象的指针(`ptr`)和重新分配给对象的内存空间长度(`size`)。如果 `ptr` 为 0, `realloc` 与函数 `malloc` 作用就相同。如果 `size` 为 0 且 `ptr` 不为 0, 则释放对象内存。否则,如果 `ptr` 不为 0 且 `size` 大于 0, `realloc` 将试图分配新的内存块。如果无法分配新内存, `ptr` 指向的对象将保持不变。函数 `realloc` 返回指向重新分配的内存的指针或空指针。
- `goto` 语句用于改变程序控制流。程序在 `goto` 语句中指定的标号后的第一条语句继续执行。
- 标号是带有一个冒号的标识符。标号必须与引用它的 `goto` 语句位于同一个函数。
- 联合体是一种派生的数据类型,其成员共享相同的存储空间。它的成员可以为任意类型。为联合体所保留的空间应该足够大到可以容纳其最大的成员。大多数情况下,联合体只包含两种或更多的数据类型。一次只能引用一个成员,也就是一种数据类型。
- 联合体的声明格式与结构相同。
- 联合体可以只用类型为其第一个成员类型的值来初始化。
- C++ 能够通过让程序员提供“接合规约”来通知编译器一个函数是在 C 编译器中编译的,从而防止了在 C++ 编译器中对函数名编码。
- 要想通知编译器一个或多个函数是在 C 中编译的,可以编写函数原型

```
extern "C" function prototype //single function
extern "C" /*multiple functions
{
    function prototypes
}
```

- 该声明通知编译器规定函数不是在 C++ 中编译的,因而不需要对它们进行名字编码。然后这些函数能够正确地与程序接合。
- C++ 环境中通常包含标准 C 函数库,因而不需要程序员对这些函数使用接合规约。

## 本章术语

append output symbol >> 附加输出符 >>

command - line arguments 命令行参数

const 类型限定符

dynamic arrays 动态数组

event 事件

extern linkage 外部连接

extern storage class specifier extern 存储类别说明符

float suffix(f or F) 浮点型后缀(f 或 F)

floating - point exception 浮点异常

goto statement goto 语句

I/O redirection I/O 重定向

illegal instruction 非法指令

internal linkage 内部连接

interrupt 中断

long double suffix(l or L) 长双精度型后缀(l 或 L)

long integer suffix (l or L) 长整型后缀(l 或 L)

pipe | 管道符|

pipng 管道输送

redirect input symbol < 重定向输入符 <

redirect output symbol > 重定向输出符 >

segmentation violation 段违规

signal - handling library 信号处理函数库

static storage class specifier static 存储类别说明符

trap 捕捉

union 联合体

unsigned integer suffix(u or U)

无符号整数后缀(u 或 U)

unsigned long integer suffix(ul or UL)

无符号长整型整数后缀(ul 或 UL)

variable-length argument list 变长参数列表

## 常见编程错误

- 18.1 将省略号放在函数参数列表中间。省略号只能放在参数列表最后。
- 18.2 引用非最后一个存储的联合体成员时会产生不确定的结果。这会将存储数据视为另一种类型。
- 18.3 比较联合体会导致语法错误,因为编译器不知道联合体的当前成员是哪个,从而不知道比较哪两个成员。
- 18.4 在联合体声明中用其类型不同于联合体第一个成员的值或表达式来初始化联合体。

## 性能提示

- 18.1 全局变量能够提高性能,因为它们能够被所有函数直接访问,消除了将数据传递给函数的开销。
- 18.2 goto 语句能用于高效跳出嵌套级别很深的控制结构。
- 18.3 使用联合体可节省存储空间。

## 可移植性提示

- 18.1 有些系统不支持长度超过 6 个字符的全局变量名或函数名。编写应用于多平台的程序时应该考虑到这一点。
- 18.2 如果数据在联合体以一种类型进行存储而以另一种类型被引用,会产生与实现过程有关的结果。
- 18.3 存储一个联合体所需的存储空间依赖于系统实现。
- 18.4 有些联合体可能不能很好地移植到其他计算机系统中。联合体是否能够被移植依赖于在给定系统中,联合体成员的数据类型的存储对齐要求。

## 软件工程知识

- 18.1 尽量避免使用全局变量,除非应用程序的性能是关键因素,因为它们违背了最低权限原则加大了软件维护的难度。
- 18.2 创建多个源文件的程序有助于软件重用和良好的软件工程实施。函数可能为多个应用程序所共同使用。在这种情况下,这些函数应该存放在它们自己的源文件中。每个源文件都有其相应的包含函数原型的头文件。这样一来,不同应用程序的程序员就能够重复使用同样的源代码,只要他们在程序中包含恰当的头文件,并将自己的应用程序和相应的源文件一起编译。
- 18.3 goto 语句应该只用于追求性能的应用程序中。它是非结构化的,可能会导致程序难于调试、维护和修改。
- 18.4 和结构或类声明一样,联合体声明只是创建了一种新的类型。将联合体或结构声明放在任何函数之外并不会创建全局变量。

## 自测题

## 18.1 填空题:

- a) 符号\_\_\_\_\_将键盘输入重定向为文件输入。
- b) 符号\_\_\_\_\_用来将屏幕输出重定向到文件。
- c) 符号\_\_\_\_\_用于把程序输出附加到文件结尾。
- d) \_\_\_\_\_用于将一个程序的输出重定向后为另一个程序的输入。
- e) 函数参数列表中的\_\_\_\_\_表明该函数能够接收可变个数的参数。
- f) 在访问可变长参数列表中的参数之前,必须调用宏\_\_\_\_\_。
- g) 宏\_\_\_\_\_用于访问可变长参数列表中的单个参数。
- h) 宏\_\_\_\_\_有助于其参数列表被宏 `va_start` 引用的函数能够正常返回。
- i) 函数 `main` 的参数\_\_\_\_\_接收命令行中参数的个数。
- j) 函数 `main` 的参数\_\_\_\_\_将命令行中的参数存储为字符串。
- k) UNIX 程序\_\_\_\_\_用来读取名为\_\_\_\_\_的其中包含编译和连接多个源文件程序所需指令的文件。这个程序只重新编译有改动的文件自上次编译以来被修改过的文件进行重新编译。
- l) 函数\_\_\_\_\_强迫程序终止执行。
- m) 函数\_\_\_\_\_注册程序正常终止时调用的函数。
- n) 类型限定符\_\_\_\_\_表示对象被初始化后就不能修改。
- o) 整型或浮点型\_\_\_\_\_能够加在一个整数或浮点数常量之后,用于指定常量的准确类型。
- p) 函数\_\_\_\_\_用于注册捕捉意外事件的函数。
- q) 函数\_\_\_\_\_可产生来自于程序内部的信号。
- r) 函数\_\_\_\_\_为数组动态分配内存,并将数组元素初始化为0。
- s) 函数\_\_\_\_\_用于改变动态分配内存的大小。
- t) \_\_\_\_\_是一个类,它包含了在不同时间内占用相同内存空间的一些变量的集合。
- u) 关键字\_\_\_\_\_用于引入联合体定义。

## 自测题答案

- 18.1 a) 重定向输入符(`<`) b) 重定向输出符(`>`) c) 附加输出符(`>>`) d) 管道符(`|`)  
 e) 省略号(`...`) f) `va_start` g) `va_arg` h) `va_end` i) `argc` j) `argv` k) `make`, `makefile` l) `exit` m) `atexit` n) `const` o) 后缀 p) `signal` q) `raise` r) `callocs` `realloc`  
 t) 联合体 u) `union`

## 练习题

- 18.2 编写程序,计算几个整数的乘积,使用变长参数列表将这些整数传递给函数 `product`。  
 每次以不同个数目的参数调用函数 `product` 测试程序。
- 18.3 编写程序,打印程序中的命令行参数。

- 18.4 编写程序,按升序或降序排序一个整数数值。程序应用命令行参数来传递表示升序的参数 -a 或表示降序的参数 -d。(注意:这是 UNIX 中将选项传递给程序的标准格式。)
- 18.5 阅读系统使用手册以确定信号处理函数库(csignal)支持哪些信号。编写程序,用信号处理器处理信号 SIGABRT 和 SIGINT。程序应该调用函数 abort 产生 SIGABRT 型信号和按 Ctrl-c 产生 SIGINT 型信号测试出对这些信号的捕捉情况。
- 18.6 编写程序,为一个整数数组动态分配内存。数组的大小从键盘输入,其元素通过键盘输入进行赋值。打印数组中的元素值。然后将数组的大小调为原来的一半。打印数组剩余的值确定它们与原来数组的值相同。
- 18.7 编写将两个文件名作为命令行参数的程序,该程序从第一个文件中逐个读取字符,再以相反的顺序将它们写入第二个文件中。
- 18.8 编写程序,用 goto 语句模拟嵌套循环结构打印如下所示的星号长方形。

```

* * * * *
*       *
*       *
*       *
*       *
* * * * *

```

程序应只用输出语句

```

cout << '*';
cout << '\n';
cout << endl;

```

- 18.9 定义一个联合体 union Data,其成员包括 char c,short s,long l,float f 和 double d。
- 18.10 创建联合体 union Integer,其成员为 char c,short s,int i 和 long l。编写程序,输入类型分别为 char,short,int 和 long 的值,并将它们存储在 union Integer 类型的一些联合体变量中。每一个联合体变量都要以 char,short,int 和 long 值打印出来。这些值总能正确打印吗?
- 18.11 创建联合体 union FloatingPoint,其成员为 float f,double d 和 long double l。编写程序,输入类型分别为 float,double 和 long double 的值,并将它们存储在 union FloatingPoint 类型的一些联合体变量中。每一个联合体变量都要以 float,double 和 long double 值打印出来。这些值总能正确打印吗?
- 18.12 针对联合体

```

union A {
    double y;
    char *z;
};

```

下列哪一条语句能正确初始化它?

- a) A p = B; //B is of same type as A
- b) A q = x; //x is a double

```
c) A r = 3.14159;  
d) A s = { 79.63 };  
e) A t = { "Hi There!" };  
f) A u = { 3.14159 , "Pi" };
```

## 第 19 章 string 类和字符串流处理

### 学习目标

- 用 C++ 标准函数库中的 string 类将字符串当作成熟的对象
- 能赋值、连接、比较、查找和交换字符串
- 能确定字符串的属性
- 能在字符串中查找、替换和插入字符
- 能将字符串转换为 C 风格的字符串
- 能字符串迭代器
- 能在内存中输入/输出字符串

### 19.1 简介

C++ 模板类 `basic_string` 提供了几种典型的字符串操作如复制和查找字符串等。类模板定义及所有支持功能都是在 `namespace std` 中定义的,其中包括的 `typedef` 语句

```
typedef basic_string< char > string;
```

为 `basic_string< char >` 创建了别名类型 `string`。同时也对类型 `wchar_t` 使用了 `typedef` 语句。类型 `wchar_t` 存储支持其他字符集的字符(如两字节的字符、4 字节的字符等)。本章只用 `string`。要想使用 `string`,必须包含标准函数库头文件 `<string>` (注意:类型 `wchar_t` 通常用于表示字符为 16 位的 Unicode 码,但标准中并没有规定类型 `wchar_t` 的大小。)

字符串对象通常用一个或二个构造函数参数来初始化。如

```
string s1("Hello"); //create string from const char *
```

就用一个参数创建了一个包含了“Hello”符(可能不包括终止符“\0”)的字符串。成用两个构造函数参数来初始化,如下所示

```
string s2(8, 'x'); //string of 8 'x' character
```

用两个参数创建了包含 8 个 'x' 字符的字符串。`string` 类提供了默认的构造函数和复制的构造函数。

字符串也可以在其定义中用另一种构造语句

```
string month = "March"; //same as: string month("March");
```

初始化。记住这里的操作符 `=` 并不是赋值操作符。它只是调用 `string` 类的构造函数隐式执行这种转换。

注意 `string` 类在其定义中并没有提供类型 `int` 或 `char` 转换为字符串的过程。例如定义

```
string error1 = 'c';  
string error2 ('u');  
string error3 = 22;  
string error4 ( 8 );
```



就会导致语法错误。同时也要注意,可在赋值语句中把单个字符赋给一个字符串对象,例如

```
s = 'n';
```

**常见编程错误 19.1** 试图通过声明中的赋值或构造函数参数将类型 `int` 或 `char` 转换为 `string` 是语法错误。

**常见编程错误 19.2** 创建一个过长而无法表示的字符串会导致系统抛出 `length_error` 异常。

与 C 风格的 `char *` 字符串不同, C++ 中的字符串并不一定要以 `null` 结尾。字符串的长度存储在字符串对象中,并且能够用成员函数 `length` 来获取。字符串也可使用下标操作符 `[]` 访问单个字符。与 C 风格的字符串类似,字符串的第一个下标为 0 最后一个下标为 `length - 1`。注意字符串并非指针,也就是说当 `s` 为字符串时,表达式 `&s[0]` 并不等于 `s`。

大多数字符串成员函数都将起始下标位置 and 要操作的字符数作为参数。

试图向 `string` 成员函数传递一个大于要处理字符数的数值,会令该数值被重新设置为要处理字符数与这个数值的差值。例如,在某函数操作长度为 50 的字符串时,如果向该函数传递 2(起始下标位置)和 100(字符数),就会产生 48(50 与 2)的差值字符数。

重载流读取操作符(`>>`)以支持字符串。语句

```
string stringObject;
cin >> stringObject;
```

从标准输入设备中读取一个字符串。输入内容用空白字符分开。头文件 `<string>` 中的函数 `getline` 也被重载以支持字符串。语句

```
string s;
getline( cin, s );
```

从键盘读取字符串到 `s` 中。输入内容用换行符(`'\n'`)分开。

## 19.2 字符串的赋值与拼接

图 19.1 中的程序演示了字符串的赋值与拼接。

```
1 //Fig.19.1: fig19_01.cpp
2 //Demonstrating string assignment and concatenation
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     string s1( "cat" ), s2, s3;
15
16     s2 = s1;           //assign s1 to s2 with =
17     s3.assign( s1 );   //assign s1 to s3 with assign()
```

```

18     cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: "
19         << s3 << "\n\n";
20
21     //modify s2 and s3
22     s2[ 0 ] = s3[ 2 ] = 'r';
23
24     cout << "After modification of s2 and s3:\n"
25         << "s1: " << s1 << "\ns2: " << s2 << "\ns3: ";
26
27     //demonstrating member function at()
28     int len = s3.length();
29     for ( int x = 0; x < len; ++x )
30         cout << s3.at( x );
31
32     //concatenation
33     string s4( s1 + "apult" ), s5; //declare s4 and s5
34
35     //overloaded +=
36     s3 += "pet"; //create "carpet"
37     s1.append( "acomb" ); //create "catacomb"
38
39     //append subscript locations 4 thru the end of s1 to
40     //create the string "comb" (s5 was initially empty)
41     s5.append( s1, 4, s1.size() );
42
43     cout << "\n\nAfter concatcenation:\n" << "s1: " << s1
44         << "\ns2: " << s2 << "\ns3: " << s3 << "\ns4: " << s4
45         << "\ns5: " << s5 << endl;
46
47     return 0;
48 }

```

输出结果:

```

s1: cat
s2: cat
s3: cat

```

After modification of s2 and s3;

```

s1: cat
s2: rat
s3: car

```

After concatenation:

```

s1: catacomb
s2: rat
s3: carpet
s4: catapult
s5: comb

```

图 19.1 字符串的赋值与拼接示例

为了使用 `string` 类,第 8 行中包含了头文件 `string`。第 14 行创建了 3 个字符串 `s1`,`s2` 和 `s3`。第 16 行

```
s2 = s1;           //assign s1 to s2 with =
```

将字符串 `s1` 赋给 `s2`。赋值后,`s2` 便是 `s1` 的副本,但 `s2` 与 `s1` 没有任何联系。第 17 行

```
s3.assign( s1 );   //assign s1 to s3 with assign( )
```

用成员函数将 `s1` 复制到 `s3`,新建了另一个副本(也就是说,`s1` 与 `s3` 是相互独立的对象)。`string` 类也提供了函数 `assign` 的重载版本用于复制指定数目的字符,例如:

```
myString.assign( s, start, numberOfChars );
```

`s` 是要复制的字符串,`start` 为起始下标,`numberOfChars` 为要复制的字符数。

第 22 行

```
s2[ 0 ] = s3[ 2 ] = 'r';
```

用下标操作符将 `'r'` 赋给 `s3[2]`(形成 `"car"`)与 `s2[0]`(形成 `"rat"`)。然后输出这些字符串。

第 28~30 行

```
int len = s3.length( );
for ( int x = 0; x < len; ++x)
    cout << s3.at( x );
```

利用 `for` 循环通过函数 `at` 的使用逐个输出 `s3` 中的字符。函数 `at` 提供了“访问检查”(或“范围检查”),也就是说,超过字符串边界时会抛出 `out_of_range` 异常(对于异常处理的详细讨论,参见第 13 章)。注意下标操作符 `[]` 并不提供这种访问检查。这与数组的用法是一致的。

**常见编程错误 19.3** 用函数 `at` 访问超出字符串边界的下标时会抛出 `out_of_range` 异常。

**常见编程错误 19.4** 用下标操作符来访问超过字符串长度的元素是逻辑错误。

第 33 行声明了字符串 `s4`,然后用重载的加号(`+`)将字符串 `s1` 与 `"apult"` 连接后的结果赋给字符串 `s4`。第 36 行

```
s3 += " carpet ";   //create "carpet"
```

利用加法赋值操作符 `+=` 连接 `s3` 与 `"pet"`。

第 37 行

```
s1.append( "acomb" ); //create "catacomb"
```

用函数 `append` 连接 `s1` 与 `"acomb"`。第 41 行

```
s5.append( s1, 4, s1.size() );
```

将从 `s1` 中的字符附加到 `s5` 中。`s1` 中的第 4 个字符到末尾的最后一个字符都连接到 `s5` 中。函数 `size` 返回字符串 `s1` 中的字符数。

## 19.3 比较字符串

`string` 类提供了比较字符串的函数。图 19.2 中的程序演示了 `string` 类的比较功能。

```
1 //Fig. 19.2: fig19_02.cpp
2 //Demonstrating string comparison capabilities
3 #include <iostream>
```

```
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     string s1( "Testing the comparison functions." ),
15             s2( "Hello" ), s3( "stinger" ), z1( s2 );
16
17     cout << "s1: " << s1 << "\ns2: " << s2
18          << "\ns3: " << s3 << "\nz1: " << z1 << "\n\n";
19
20     //comparing s1 and z1
21     if ( s1 == z1 )
22         cout << "s1 == z1\n";
23     else { //s1 != z1
24         if ( s1 > z1 )
25             cout << "s1 > z1\n";
26         else //s1 < z1
27             cout << "s1 < z1\n";
28     }
29
30     //comparing s1 and s2
31     int f = s1.compare( s2 );
32
33     if ( f == 0 )
34         cout << "s1.compare( s2 ) == 0\n";
35     else if ( f > 0 )
36         cout << "s1.compare( s2 ) > 0\n";
37     else //f < 0
38         cout << "s1.compare( s2 ) < 0\n";
39
40     //comparing s1 (elements 2 - 5) and s3 (elements 0 - 5)
41     f = s1.compare( 2, 5, s3, 0, 5 );
42
43     if ( f == 0 )
44         cout << "s1.compare( 2, 5, s3, 0, 5 ) == 0\n";
45     else if ( f > 0 )
46         cout << "s1.compare( 2, 5, s3, 0, 5 ) > 0\n";
47     else //f < 0
48         cout << "s1.compare( 2, 5, s3, 0, 5 ) < 0\n";
49
50     //comparing s2 and z1
51     f = z1.compare( 0, s2.size(), s2 );
52
53     if ( f == 0 )
54         cout << "z1.compare( 0, s2.size(), s2 ) == 0" << endl;
```

```

55     else if ( f > 0 )
56         cout << "z1.compare( 0, s2.size(), s2 ) > 0" << endl;
57     else //f < 0
58         cout << "z1.compare( 0, s2.size(), s2 ) < 0" << endl;
59
60     return 0;
61 }

```

输出结果:

```

s1: Testing the comparison functions.
s2: Hello
s3: stinger
z1: Hello

s1 > z1
s1.compare( s2 ) > 0
s1.compare( 2, 5, s3, 0, 5 ) == 0
z1.compare( 0, s2.size(), s2 ) = 0

```

图 19.2 比较字符串

程序在第 14 行和第 15 行声明了 4 个字符串

```

string s1( "Testing the comparison functions." ),
        s2( "Hello" ), s3( "stinger" ), z1(s2);

```

并在第 17 行和第 18 行输出了它们。第 21 行中的条件

```
s1 == z1
```

用于测试 s1 与 z1 是否相等。如果条件为 true, 则输出“s1 == z1”。若为假, 便测试第 24 行中的条件

```
s1 > z1
```

这里演示的重载操作符函数以及那些没有演示的( !=, <, >= 和 <= )都返回 bool 值。

第 31 行

```
int f = s1.compare( s2 );
```

用字符串函数 compare 测试字符串 s1 与 s2 是否相等。如果已变量 f, 且 s1 与 s2 相等就会为 f 赋值为 0; 如果 s1“按词典顺序”大于 s2, f 就会被赋值为正数; 如 s1“按词典顺序”小于 s2, f 就会被赋值为负数。

第 41 行

```
f = s1.compare( 2, 5, s3, 0, 5 );
```

用函数 compare 的重载版本比较字符串 s1 与 s3 的部分内容。前两个参数 2 和 5 指定了 s1 与 s3 比较部分的起始下标与长度。第 3 个参数为比较字符串。最后两个参数 0 和 5 为比较字符串中被比较部分的起始下标与长度。其结果被将给 f; 如果 s1 与 s2 相等, f 赋值为 0; 如果 s1“按词典顺序”大于 s2, f 就赋为正数, 如 s1“按词典顺序”小于 s2, f 就赋为负数。

第 51 行

```
f = z1.compare( 0, s2.size(), s2 );
```

利用用函数 compare 的另一个重载版本比较 z1 与 s2。第一个参数指定了 z1 中比较部分的起始下标。第二个参数指定了 z1 中比较部分的长度。函数 size 返回指定字符串中字符数。最后一个参数为比较字符串。其结果将赋给 f; 如果 s1 与 s2 相等, 则赋值为 0; 如果 s1“按词

顺序”大于 s2,则赋值为正数,如 s1“按词典顺序”小于 s2,则赋值为负数。

## 19.4 子串

string 类提供了函数 substr,用于在一个字符串中读取子串。图 19.3 中的程序演示了函数 substr 的用法。

```

1 //Fig.19.3: fig19_03.cpp
2 //Demonstrating function substr
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     string s( "The airplane flew away." );
15
16     //retrieve the substring "plane" which
17     //begins at subscript 7 and consists of 5 elements
18     cout <<s.substr( 7, 5 ) <<endl;
19
20     return 0;
21 }
```

输出结果:

plane

图 19.3 函数 substr 用法示例

程序在第 18 行声明并初始化了一个字符串。下一行语句

```
cout <<s.substr( 7, 5 ) <<endl;
```

利用函数 substr 从 s 中取出一个子串。第一个参数指定了子串的起始下标。最后一个参数指定了要读取的字符。

## 19.5 交换字符串

string 类提供了函数 swap,用于交换字符串。图 19.4 中的程序演示了如何交换两个字符串。

```

1 //Fig.19.4: fig19_04.cpp
2 //Using the swap function to swap two strings
3 #include <iostream>
4
5 using std::cout;
```

```
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 |
14     string first( "one" ), second( "two" );
15
16     cout << "Before swap:\n first: " << first
17         << "\nsecond: " << second;
18     first.swap( second );
19     cout << "\n\nAfter swap:\n first: " << first
20         << "\nsecond: " << second << endl;
21
22     return 0;
23 |
```

输出结果:

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

图 19.4 用函数 swap 交换两个字符串

第 14 行声明并初始化了两个字符串 first 与 second。然后输出它们。第 18 行

```
first.swap( second);
```

用函数 swap 交换 first 与 second 的值。然后再次打印这两个字符串,确定它们确实进行了交换。

## 19.6 字符串的特性

string 类提供了用于统计字符串大小、长度、容量、最大长度和其他一些特性的函数。字符串的“大小”或“长度”为当前存储在字符串中的字符数。字符串的“容量”是在不增加内存的情况下能够存储在字符串中元素的总个数。“最大长度”为能够存储在字符串对象中字符串的最大长度。图 19.5 中的程序演示了如何用 string 类函数确定字符串的大小、长度和其他。

```
1 //Fig. 19.5: fig19_05.cpp
2 //Demonstrating functions related to size and capacity
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::cin;
```

```
8
9 #include <string>
10
11 using std::string;
12
13 void printStats( const string & );
14
15 int main()
16 {
17     string s;
18
19     cout << "Stats before input:\n";
20     printStats( s );
21
22     cout << "\n\nEnter a string: ";
23     cin >> s; //delimited by whitespace
24     cout << "The string entered was: " << s;
25
26     cout << "\nStats after input:\n";
27     printStats( s );
28
29     s.resize( s.length() + 10 );
30     cout << "\n\nStats after resizing by (length + 10):\n";
31     printStats( s );
32
33     cout << endl;
34     return 0;
35 }
36
37 void printStats( const string &str )
38 {
39     cout << "capacity: " << str.capacity()
40         << "\nmax size: " << str.max_size()
41         << "\nsize: " << str.size()
42         << "\nlength: " << str.length()
43         << "\nempty: " << ( str.empty() ? "true": "false" );
44 }
```

**输出结果:**

```
Stats before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Stats after input:
capacity: 31
max size: 4294967293
size: 6
```



```
length: 6
empty: false

Stats after resizing by (length + 10):
capacity: 31
max size: 4294967293
size: 16
length: 16
empty: false
```

图 19.5 打印字符串统计信息

程序声明了空字符串 `s` (第 17 行), 并将其传递给函数 `printStats` (第 20 行)。空字符串是不包含任何字符的字符串。通过键盘输入字符串“tomato”, 注意字符串是用空白字符分隔, 这样可避免输入多余的字符。

函数 `printStats` 将常量字符串的引用作为参数, 并输出字符串的容量 (使用函数 `capacity`)、最大长度 (用函数 `max_size`)、大小 (使用函数 `size`)、长度 (用函数 `length`) 和字符串是否为空 (用函数 `empty`) 等信息。初次调用 `printStats` 时 `s` 的初始容量、大小与长度值均为 0。因为 `s` 的初始容量为 0, 所以在 `s` 中加入字符时, 就应为之分配内存以容纳新字符。大小和长度为 0 表明 `s` 当前没有存储任何字符, 大小与长度总是相同的, 这里最大长度为 4 294 967 293。字符串 `s` 为空字符串, 因此函数 `empty` 返回 `true`。

第 23 行将一个字符串输入 `s`。注意, 这里使用了流读取操作符 `>>`。第 29 行

```
s.resize( s.length() + 10 );
```

用函数 `resize` 将 `s` 的长度增加了 10 个字符。

## 19.7 查找字符串中的字符

`string` 类提供了一些函数, 用于在字符串中查找字符串与字符。图 19.6 中的程序演示了“查找”函数, 所有查找函数均为常量类型。

```
1 //Fig.19.6: fig19_06.cpp
2 //Demonstrating the string find functions
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     //compiler concatenates all parts into one string literal
15     string s( "The values in any left subtree"
16              " \nare less than the value in the"
17              " \nparent node and the values in"
18              " \nany right subtree are greater"
```

```

19         " \nthan the value in the parent node" );
20
21     //find "subtree" at locations 23 and 102
22     cout << "Original string: \n" << s
23         << " \n \n(find) \"subtree\" was found at: "
24         << s.find( "subtree" )
25         << " \n \n(rfind) \"subtree\" was found at: "
26         << s.rfind( "subtree" );
27
28     //find 'p' in parent at locations 62 and 144
29     cout << " \n(find_first_of) character from \"qpxz\" at: "
30         << s.find_first_of( "qpxz" )
31         << " \n \n(find_last_of) character from \"qpxz\" at: "
32         << s.find_last_of( "qpxz" );
33
34     //find 'b' at location 25
35     cout << " \n(find_first_not_of) first character not \n"
36         << "     contained in \"heTv lusi nodrpayft\"; "
37         << s.find_first_not_of( "heTv lusi nodrpayft" );
38
39     //find 'n' at location 121
40     cout << " \n(find_last_not_of) first character not \n"
41         << "     contained in \"heTv lusi nodrpayft\"; "
42         << s.find_last_not_of( "heTv lusi nodrpayft" ) << endl;
43
44     return 0;
45 }

```

#### 输出结果:

```

Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

(find) "subtree" was found at: 23
(rfind) "subtree" was found at: 102
(find_first_of) character from "qpxa" at: 62
(find_last_of) character from "qpxz" at: 144
(find_first_not_of) first character not
    contained in "heTv lusi nodrpayft"; 25
(find_last_not_of) first character not
    contained in "heTv lusi nodrpayft"; 121

```

图 19.6 字符串查找函数用法示例

字符串 `s` 在第 15 行得以声明并初始化,编译器将这 5 个字符串连成一个字符串。为避免语法错误,在转到下一行开始下一个字符串之前,每个字符串的末尾都必须用双引号封闭起来。

**常见编程错误 19.5** 不用双引号来结束字符串是语法错误。

## 第 24 行中的插入操作

```
s.find("subtree")
```

用函数 `find`, 试图在字符串 `s` 中查找字符串“subtree”。如果找到, 函数就返回这个字符串起始位置的下标。如果找不到, 函数就返回值 `string::npos` (`string` 类中定义的一个 `public` 静态常量)。字符串相关查找函数返回的这个值表示无法在字符串中找到子串或字符。

## 第 26 行, 流插入操作的最后一项输出

```
s.rfind("subtree") //reverse find
```

它用函数 `rfind` 从后到前逆向查找字符串。如果找到要查找的字符串, 函数就返回相应的下标位置。如果找不到, 就返回 `string::npos`。(注意, 除非特别声明, 本节提到的查找函数都会返回相同的值类型。)注意, 在其他环境下, 常量 `string::npos` 也用于表示一个字符串的所有元素。

## 第 30 行

```
s.find_first_of("qpxz");
```

用函数 `find_first_of` 在字符串中查找“qpxz”中任一字符首次出现的位置。查找从 `s` 的头部开始进行。在第 62 位找到了字符“p”。

## 第 32 行

```
s.find_last_of("qpxz");
```

用函数 `find_last_of` 在字符串中查找“qpxz”中任一字符最后一次出现的位置。查找从 `s` 的末尾处开始, 在第 144 位找到字符“p”。

## 第 37 行

```
s.find_first_not_of("qpxz");
```

用函数 `find_first_not_of` 在字符串 `s` 中查找第一个不包含在字符串“heTv lusinodrpayft”中的第一个字符。查找从 `s` 的头部开始。

## 第 42 行

```
s.find_last_not_of("heTv lusinodrpayft");
```

用函数 `find_last_not_of` 来查找不包含在“heTv lusinodrpayft”中的第一个字符。查找从 `s` 末尾处开始。

## 19.8 替换字符串中的字符

图 19.7 演示了用于替换和删除字符的函数。

```
1 //Fig. 19.7: fig19_07.cpp
2 //Demonstrating functions erase and replace
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
```

```

12 int main()
13 {
14     //compiler concatenates all parts into one string
15     string s( "The values in any left subtree"
16             "are less than the value in the"
17             "parent node and the values in"
18             "any right subtree are greater"
19             "than the value in the parent node" );
20
21     //remove all characters from location 62
22     //through the end of s
23     s.erase( 62 );
24
25     //output the new string
26     cout << "Original string after erase;\n" << s
27         << "\n\nAfter first replacement;\n";
28
29     //replace all spaces with a period
30     int x = s.find( " " );
31     while ( x < string::npos ) {
32         s.replace( x, 1, "." );
33         x = s.find( " ", x + 1 );
34     }
35
36     cout << s << "\n\nAfter second replacement;\n";
37
38     //replace all periods with two semicolons
39     //NOTE: this will overwrite characters
40     x = s.find( "." );
41     while ( x < string::npos ) {
42         s.replace( x, 2, "xxxxx;yyy", 5, 2 );
43         x = s.find( ".", x + 1 );
44     }
45
46     cout << s << endl;
47     return 0;
48 }

```

输出结果:

```

Original string after erase:
The values in any left subtree
are less than the value in the

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;han;;he;;alue;;n;;he

```

图 19.7 函数 erase 与 replace 用法示例

程序首先声明并初始化了字符串 s。第 23 行

```
s.erase( 62 );
```

用函数 `erase` 删除 s 中第 62 个元素到字符串结束处的所有元素。

第 30 ~ 34 行

```
int x = s.find( " " );
while ( x < string::npos ) {
    s.replace( x, 1, "." );
    x = s.find( " ", x+1 );
}
```

用函数 `find` 查找空白字符的每次出现的位置,然后调用函数 `replace` 将空格替换为句号。函数 `replace` 带 3 个参数:起始下标、要替换的字符数和替换字符。常量 `string::npos` 代表最大的字符长度。到达 s 末尾时,函数 `find` 返回 `string::npos`。

第 40 ~ 44 行

```
x = s.find( ".");
while ( x < string::npos ) {
    s.replace( x, 2, "xxxxx;yyy", 5, 2);
    x = s.find( ".", x+1 );
}
```

用函数 `find` 查找每一个句号并用函数 `replace` 将每个句号及紧随其后的那个字符替换为两个分号。传递给 `replace` 的参数为替换操作开始的元素下标、要替换的字符数、一个替换字符串(其子串之一将用于替换字符)、替换子串的开始元素以及替换字符串中用于替换的字符数(也即替换子串的字符数)。

## 19.9 在字符串中插入字符

`string` 类提供了用于在字符串中插入字符的函数。图 19.8 中的程序演示了 `string` 类的插入功能。

```
1 //Fig.19.8: fig19_08.cpp
2 //Demonstrating the string insert functions.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 |
14     string s1( "beginning end" ),
15         s2( "middle " ), s3( "12345678" ), s4( "xx" );
16
```

```

17     cout << "Initial strings: \ns1: " << s1
18         << " \ns2: " << s2 << " \ns3: " << s3
19         << " \ns4: " << s4 << " \n\n";
20
21     //insert "middle" at location 10
22     s1.insert( 10, s2 );
23
24     //insert "xx" at location 3 in s3
25     s3.insert( 3, s4, 0, string::npos );
26
27     cout << "Strings after insert: \ns1: " << s1
28         << " \ns2: " << s2 << " \ns3: " << s3
29         << " \ns4: " << s4 << endl;
30
31     return 0;
32 }

```

输出结果:

```

Initial strings;
s1: beginning end
s2: middle
s3: 12345678
s4: xx

Strings after insert;
s1: beginning middle end
s2: middle
s3: 123xx45678
s4: xx

```

图 19.8 字符串插入函数用法示例

程序首先声明并初始化了 4 个字符串 s1, s2, s3 和 s4。然后输出它们。第 22 行

```
s1.insert( 10, s2 );
```

用函数 insert 将字符串 s2 插入 s1 的第 10 个元素之前。

第 25 行

```
s3.insert( 3, s4, 0, string::npos );
```

用 insert 将 s4 插入 s3 的第 3 个元素之前。最后两个参数指定了 s4 的起始元素以及要插入 s4 中的字符数。

**性能提示 19.1** 插入操作会导致额外的内存管理操作,从而降低性能。

## 19.10 转换为 C 风格的 char\* 字符串

string 类提供了函数用于将字符串转换为 C 风格的字符串。前面讲过,与 C 风格的字符串不同,C++ 中的字符串不一定要用空字符结束。指定函数取 C 风格的字符串作为参数时,这些转换函数便可派上用场。图 19.9 中的程序演示了如何将字符串转换为 C 风格的字符串。

```

1 //Fig.19.9: fig19_09.cpp
2 //Converting to C-style strings.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 |
14     string s( "STRINGS" );
15     const char *ptr1 = 0;
16     int len = s.length();
17     char *ptr2 = new char[ len + 1 ]; //including null
18
19     //copy characters out of string into allocated memory
20     s.copy( ptr2, len, 0 );
21     ptr2[ len ] = 0; //add null terminator
22
23     //output
24     cout << "string s is " << s
25         << " \ns converted to a C-Style string is "
26         << s.c_str() << " \nptr1 is ";
27
28     //Assign to pointer ptr1 the const char * returned by
29     //function data(). NOTE: this is a potentially dangerous
30     //assignment. If the string is modified, the pointer
31     //ptr1 can become invalid.
32     ptr1 = s.data();
33
34     for ( int k = 0; k < len; ++k )
35         cout << *( ptr1 + k ); //use pointer arithmetic
36
37     cout << " \nptr2 is " << ptr2 << endl;
38     delete [] ptr2;
39     return 0;
40 |

```

输出结果:

```

string s is STRINGS
s converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS

```

图 19.9 将字符串转换为 C 风格的字符串和字符数组

程序声明了一个字符串、一个整数和两个指针。字符串 s 初始化为“STRINGS”，ptr1 初始化为 0，len 初始化为 s 的长度，然后动态分配内存并令 ptr2 指向它。

第 20 行

```
s.copy( ptr2, len, 0);
```

用函数 copy 将 s 复制到 ptr2 指向的数组。将字符串转换为 C 风格的字符串是隐式进行的。第 21 行在数组 ptr3 中放入一个空字符终止。

第 26 行的第一个流插入操作

```
<< s.c_str( )
```

用于显示转换 s 时 c\_str 返回的以空字符终止的 const char \*。

第 32 行

```
ptr1 = s.data( );
```

将 data 返回的以非空字符结束的字符数组 const char \* 赋给 ptr1。注意本例中我们并没有修改字符串 s。如果修改了 s, ptr1 可能变得无效且导致不可预料的结果。

注意 data 返回的字符数组和 c\_str 返回的 C 风格的字符数组, 它们的存在期是有限的。它们属于 string 类而且不应该用 delete 进行删除。

第 34 行和第 35 行用指针算法输出 ptr1 指向的数组。在第 37 行和第 38 行中, 输出了 ptr2 指向的 C 风格的字符串, 然后释放分配给 ptr2 的内存以防止内存泄露。

**常见编程错误 19.6** 未用空字符结束 data 或 copy 返回的字符数组, 可能会导致执行时错误。

**良好编程习惯 19.1** 尽可能用健壮的 C++ 字符串而不是 C 风格的字符串。

**常见编程错误 19.7** 将包含一个或多个空字符的字符串转换为 C 风格的字符串可能会导致逻辑错误。因为 C 风格的字符串会把空字符解释为终止符。

## 19.11 迭代器

string 类提供了“迭代器”用于向前和向后遍历字符串。为访问单个的字符迭代器提供了与指针操作相似的语法。迭代器不检查范围。注意本节中, 我们提供了一个非常简单的例子, 演示迭代器的用法。下一章, 我们会讨论迭代器更健壮的使用法。图 19.10 中的程序演示了迭代器的用法。

```
1 //Fig.19.10: fig19_10.cpp
2 //Using an iterator to output a string.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     string s( "Testing iterators" );
```



```

15     string::const_iterator i1 = s.begin();
16
17     cout << "s = " << s
18         << "\n(Using iterator i1) s is: ";
19
20     while ( i1 != s.end() ) {
21         cout << *i1;    //dereference iterator to get char
22         ++i1;           //advance iterator to next char
23     }
24
25     cout << endl;
26     return 0;
27 }

```

输出结果:

```

s = Testing iterators
(Using iterator i1) s is: Testing iterators

```

图 19.10 利用迭代器输出字符串

#### 第 14 行和第 15 行

```

string s = "Testing iterators";
string::const_iterator i1 = s.begin();

```

声明了字符串 `s` 与 `string::const_iterator i1`。`const_iterator` 是一个迭代器,不能修改它所迭代的容器(在这里为字符串)。然后用 `string` 类的函数 `begin` 将迭代器 `i1` 初始化为 `s` 的开始位置。函数 `begin` 有两个版本:一个版本返回迭代非常量字符串的迭代器;另一个常量版本返回迭代常量字符串的迭代器 `const_iterator`。然后输出字符串 `s`。

#### 第 20 ~ 23 行

```

while ( i1 != s.end() ) {
    cout << * i1;
    ++i1;
}

```

用迭代器 `i1` 遍历 `s`。函数 `end` 返回 `s` 最后一个元素之后第一个位置的迭代器。每个位置的内容通过反引用迭代器(就像反引用指针一样)打印出来,然后用操作符 `++` 将迭代器向前移一位。

`string` 类提供了成员函数 `rend` 与 `rbegin`,从字符串尾到字符串头逐个访问字符串中的字符。成员函数 `rend` 与 `rbegin` 可能返回 `reverse_iterator` 和 `const_reverse_iterator`(基于字符串是否为常量)。我们将让大家在练习中演示这些函数。同时还将在第 20 章更多地使用迭代器与逆向迭代器。

**测试和调试提示 19.1** 需要范围检查时,应用字符串成员函数 `at`(而不是迭代器)。

## 19.12 字符串流处理

除了标准流 I/O 和文件流 I/O 外,C++ 流 I/O 还能从内存中的字符串获取输入和输出到内存中的字符串。这些功能通常称为“内存内 I/O”或“字符串流处理”。

istringstream 类支持从字符串获取输入,ostreamstream 类支持输出到字符串。类名 istringstream 与 ostreamstream 实际上是别名。这些名字通过 typedef 来定义,如下所示

```
typedef basic_istringstream < char > istringstream;
typedef basic_ostreamstream < char > ostreamstream;
```

除了提供与 istringstream 和 ostreamstream 类相同的功能外, basic\_istringstream 和 basic\_ostreamstream 类还提供了其他指定用于内存格式化的成员函数。使用了内存格式化的程序必须包含头文件 <sstream> 与 <iostream>。

这些技术的应用之一便是数据验证。一个程序能从输入流中一次将一整行数据读入字符串。接着,一个验证程序会检查字符串中的内容并在必要的时候纠正(或修复)数据。然后,程序在知道数据已以正确格式输入的情况下继续从字符串中获取输入。

要想充分利用 C++ 流强大的输出格式化功能,最好的方式无疑 输出到字符串。可以将数据放在字符串,模拟经过编辑的屏幕格式。字符串能够写入磁盘,保存屏幕映像。

ostreamstream 对象用字符串对象存储要输出的数据,它的成员函数 str 返回字符串的一个副本。

图 19.11 演示了 ostreamstream 对象。程序首先创建了 ostreamstream 对象 outputString(第 18 行)并用流插入操作符输出一系列的字符串或对象的数字值。

```
1 //Fig. 19.11: fig19_11.cpp
2 //Using a dynamically allocated ostreamstream object.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 #include <sstream>
13
14 using std::ostreamstream;
15
16 int main()
17 {
18     ostreamstream outputString;
19     string s1( "Output of several data types " ),
20           s2( "to an ostreamstream object:" ),
21           s3( "\n          double: " ),
22           s4( "\n          int: " ),
23           s5( "\naddress of int: " );
24     double d = 123.4567;
25     int i = 22;
26
27     outputString << s1 << s2 << s3 << d << s4 << i << s5 << &i;
28     cout << "outputString contains: \n" << outputString.str();
29 }
```

```

30     outputString << "\nmore characters added";
31     cout << "\n\nafter additional stream insertions, \n"
32         << "outputString contains: \n" << outputString.str()
33         << endl;
34
35     return 0;
36 }

```

输出结果:

```

outputString contains;
Output of several data types to an ostringstream object;
    double: 123.457
    int: 22
address of int: 0068FD0C
after additional stream insertions,
outputString contains;
Output of several data types to an ostringstream object;
    double: 123.457
    int: 22
address of int: 0068FD0C
more characters added

```

图 19.11 使用动态分配的 `ostringstream` 对象

### 第 27 行

```
output << s1 << s2 << s3 << d << s4 << i << s5 << &i;
```

将字符串 `s1`、字符串 `s2`、字符串 `s3`、双精度值 `d`、字符串 `s4`、整型值 `i`、字符串 `s5` 或整型值 `i` 的地址全部输出到内存中的 `outputString`。

### 第 28 行

```
cout << "outputString contains: \n" << outputString.str();
```

调用 `outputString.str()` 输入第 27 行中创建的字符串副本。第 30 行演示了如何只用流插入操作符而将更多数据追加到内存中的字符串中。追加了一些字符后,第 32 行输出字符串 `outputString`。

`istringstream` 对象从内存中的字符串将数据输入到程序变量,数据以字符的形式存储在 `istringstream` 对象中。从 `istringstream` 对象输入与一般的文件输入或从标准输入设备输入是一样的。`istringstream` 对象将字符串尾解释为文件尾。

图 19.12 中的程序演示了从 `istringstream` 对象获取输入。

```

1 //Fig.19.12; fig19_12.cpp
2 //Demonstrating input from an istringstream object.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11

```

```

12 #include <sstream>
13
14 using std::istringstream;
15
16 int main()
17 {
18     string input( "Input test 123 4.7 A" );
19     istringstream inputString( input );
20     string string1, string2;
21     int i;
22     double d;
23     char c;
24
25     inputString >> string1 >> string2 >> i >> d >> c;
26
27     cout << "The following items were extracted\n"
28         << "from the istringstream object;"
29         << "\nstring: " << string1
30         << "\nstring: " << string2
31         << "\n  int: " << i
32         << "\ndouble: " << d
33         << "\n char: " << c;
34
35     //attempt to read from empty stream
36     long x;
37
38     inputString >> x;
39
40     if ( inputString.good() )
41         cout << "\n\nlong value is: " << x << endl;
42     else
43         cout << "\n\ninputString is empty" << endl;
44
45     return 0;
46 }

```

输出结果:

```

The following items were extracted
from the istringstream object;
String: Input
String: test
  int: 123
double: 4.7
  char: A
inputString is empty

```

图 19.12 利用 istringstream 对象获取输入

第 18 行和第 19 行

```

string input( "Input test 123 4.7 A");
istringstream inputString( input );

```

创建了包含数据的字符串和用于包含字符串 input 中数据的 istream 对象。字符串 input 包含数据

```
Input test 123 4.7 A
```

当这些数据作为输入读取到程序中时将包含两个字符串(“Input”和“test”)、一个整型值(123)、一个双精度值(4.7)和一个字符(‘A’)。第 25 行中的数据

```
inputString >> string1 >> string2 >> i >> d >> c;
```

分别被读取到变量 string1, string2, i, d 和 c 中。然后在第 27 行到第 33 行输出这些数据。第 38 行, 程序试图再次从 inputString 读取数据。因为, 没有数据, 所以 if 条件(第 40 行)计算结果为 false, 程序转而执行 if/else 结构中的 else 部分。

## 19.13 小结

- 模板类 basic\_string 提供了基本的字符串处理操作如复制、查找等。
- typedef 语句

```
typedef basic_string<char> string;
```

创建了为 basic\_string<char> 类型 string。同时也为类型 wchar\_t 提供了 wchar\_t 类型。类型 wchar\_t 存储支持其他字符集的字符。但标准中没有规定 wchar\_t 的长度。

- 要使用 string(类或对象), 必须包含 C++ 标准头文件 <string>。
- string 类没有提供 int 或 char 到 string 的转换。
- 在赋值语句中, 可将单个字符赋给字符串对象。
- 字符串不一定要以空字符结束。
- 字符串的长度存储在字符串对象中, 且能用其成员函数 length 或 size 读取。
- 大多数字符串成员函数将起始位置下标和要操作的字符数作为参数。
- 试图向 string 成员函数传递大于要处理字符数的数值, 该值就会被重新设置为要处理字符数与该值的差值。
- string 类提供重载操作符 = 和成员函数 assign 为字符串赋值。
- 下标操作符[] 可用于直接访问字符串中的元素。
- 函数 at 提供了访问检查, 超出字符串边界时会抛出 out\_of\_range 异常。下标操作符没有提供这种访问检查。
- string 类提供重载的操作符 + 与 = 以及成员函数 append 用于连接字符串。
- string 类提供重载的操作符 ==, !=, <, >, <= 和 >= 用于比较字符串。
- 字符串函数 compare 比较两个字符串(或子串)。如果两者相等, 函数返回 0; 如果第一个字符串“按词典顺序”大于第二个字符串, 则返回正数; 如果第一个字符串“按词典顺序”小于第二个字符串, 则返回负数。
- 函数 substr 在字符串中取出一个子串。
- 函数 swap 用于交换两个字符串中的内容。
- 函数 size 和 length 返回一个字符串的大小或长度(即存储在字符串中的字符数)。
- 函数 capacity 返回在不增加内存的情况下能够存储在字符串中的元素的总个数。

- 函数 `max_size` 返回字符串能够存储的最大长度。
- 函数 `resize` 用于改变字符串的长度。
- `string` 类的查找函数 `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of` 和 `find_last_not_of` 用于查找字符串中的字符串或字符。
- 值 `string::npos` 通常用来表示要处理字符的函数中处理 `string` 的所有元素。
- 函数 `erase` 用于删除字符串中的元素。
- 函数 `replace` 用于在字符串中替换字符。
- 函数 `insert` 用于在字符串插入字符。
- 函数 `c_str` 返回 `const char *`, 空字符终止的 C 风格字符串, 该字符串包含字符串中的所有字符。
- 函数 `data` 返回一个 `const char *`, 指向非空字符终止的 C 风格字符数组, 该数组中包含字符串中的所有元素。
- `string` 类提供了成员函数 `end` 与 `begin` 用于逐个访问字符。
- `string` 类提供了成员函数 `rend` 与 `rbegin` 用于从后到前逆向访问各个字符。
- 类型 `istringstream` 支持从字符串获取输入。类型 `ostringstream` 支持输出到字符串。
- 使用内存格式化的程序必须包含头文件 `<sstream>` 与 `<iostream>`。
- `ostringstream` 成员函数 `str` 返回一个字符串的副本。

## 本章术语

access function access 函数

at function at 函数

capacity function capacity 函数

capacity 容量

checked access 访问检查

compare function compare 函数

copy function copy 函数

`c_str` function `c_str` 函数

empty string 空字符串

equality operators: `=`, `!=` 等值操作符 `=` 与 `!=`

erase function erase 函数

find function find 函数

`find_first_not_of` function `find_first_not_of` 函数

`find_first_of` function `find_first_of` 函数

`find_last_not_of` function `find_last_not_of` 函数

`find_last_of` function `find_last_of` 函数

"find" function 查找函数

getline function getline 函数

in-core I/O 内存 I/O

in-memory I/O 内存 I/O

insert function insert 函数

`istringstream` class `istringstream` 类

iterator 迭代器

length function length 函数

length of a string 字符串的长度

`length_error` exception `length_error` 异常

maximum size of a string 字符串的最大长度

`max_size` function `max_size` 函数

operators: `+`, `+=`, `<<`, `>>`, `[]`

操作符 `+`, `+=`, `<<`, `>>` 和 `[]`

`ostringstream` class `ostringstream` 类

`out_of_range` exception `out_of_range` 异常

`range_error` exception `range_error` 异常

relational operators: `>`, `<`, `>=`, `<=`

关系操作符 `>`, `<`, `>=`, `<=`

`rend` function `rend` 函数

`replace` function `replace` 函数

`resize` function `resize` 函数

`reverse_iterator` 迭代器

`size` function `size` 函数

`<string>` header file `<string>` 头文件

`str` `string-stream` member function

字符串流成员函数 `str`

`string` class `string` 类

subscript operator `[]` 下标操作符 `[]`

`substr` function `substr` 函数

swap function swap 函数

&lt;sstream&gt; header file &lt;sstream&gt; 头文件

wchar\_t type wchar\_t 类型

wide characters 宽位字符

## 常见编程错误

- 19.1 试图通过声明中的赋值或构造函数参数,将类型 int 或 char 转换为 string 是语法错误。
- 19.2 创建一个过长而无法表示的字符串会导致系统抛出 length\_error 异常。
- 19.3 使用函数 at 访问超出字符串边界的下标会导致抛出 out\_of\_range 异常。
- 19.4 使用下标操作符来访问字符串范围之外的元素会导致逻辑错误。
- 19.5 不用双引号来终止字符串是语法错误。
- 19.6 不用空字符终止 data 或 copy 返回的字符数组可能会导致执行时错误。
- 19.7 将包含一个或多个空字符的字符串转换为 C 风格的字符串可能会导致逻辑错误。因为 C 风格的字符串会将空字符解释为终止符。

## 良好编程习惯

- 19.1 尽可能用健壮的 C++ 字符串,而不是 C 风格的字符串。

## 性能提示

- 19.1 插入操作会导致额外的内存管理操作,从而降低性能。

## 测试和调试提示

- 19.1 需要范围检查时,应用字符串成员函数 at(而不是迭代器)。

## 自测题

### 19.1 填空题:

- a) 为使用 string 类,必须包含头文件\_\_\_\_\_。
- b) string 类属于\_\_\_\_\_名称空间。
- c) 函数\_\_\_\_\_用来删除字符串中的字符。
- d) 函数\_\_\_\_\_查找一串字符中任意一个字符的第一次出现。

### 19.2 判断正误。如果有错,请说明原因。

- a) 可用加法操作符 += 连接字符串。
- b) 字符串中的字符从元素 0 开始。
- c) 赋值操作符 = 复制一个字符串。
- d) C 风格的字符串属于字符串对象。

### 19.3 指出下列语句中的错误,并说明如何改正。

- a) 

```
string sv( 28 ); //construct sv
    string bc('z'); ///construct bc
```
- b) 

```
//assume std namespace is known
const char *ptr = name.data( ); //name is "joe bob"
ptr[3] = '-';
```

```
cout <<ptr <<endl;
```

### 自测题答案

- 19.1 a) <string> b) std c) erase d) find\_first\_of
- 19.2 a) 正确。  
 b) 正确。  
 c) 正确。  
 d) 正确。因为字符串对象提供许多不同的服务。C 风格的字符串不提供任何服务。C 风格的字符串以空字符结束,字符串对象却不然。
- 19.3 a) 用于被传递参数的构造函数并不存在。必要时,应该用其他构造函数将被传递的参数转换为字符串。  
 b) 函数 data 并不增加空字符。而是用函数 c\_str 增加。

### 练习题

#### 19.4 填空题:

- a) 函数\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_将字符串转换为 C 风格的字符串。  
 b) 函数\_\_\_\_\_用于赋值。  
 c) 函数 rbegin 的返回值类型为\_\_\_\_\_。  
 d) 函数\_\_\_\_\_用于取出子串。

#### 19.5 判断正误。如果有错,请说明原因。

- a) 字符串对象以空字符结尾。  
 b) 函数 max\_size 返回一个字符串的最大长度。  
 c) 函数 at 能够抛出 out\_of\_range 异常。  
 d) 函数 begin 返回一个迭代器。  
 e) 字符串对象默认按引用传递。

#### 19.6 找出下列语句中的错误,并说明如何改正。

- a) `std::cout <<s.data() <<std::endl; //s is "hello"`  
 b) `erase( s.rfind('x'),1); //s is "xenon"`  
 c) `string& foo( void )`  
`{`  
`string s("hello");`  
`... //other statement of function`  
`return;`  
`}`

#### 19.7 (简单加密法)Internet 上有些信息采用“rot13”加密算法。这种算法将每个字母按字母顺序循环 13 位。这也就是说,'a'对应于'n','x'就对应于'k'。rot13 属于对称密钥加密算法。在对称密钥加密算法中,加密与解密采用的密钥。

- a) 编写程序,用 rot13 加密一条信息。  
 b) 编写程序,用密钥 13 解密已加密的信息。



c) 完成了上面两部分的程序后。回答下面问题:

如果不知道 b) 部分的密钥, 用现有资源来破解会有多大难度? 如果能用高速运算的资源(如 cray 超级计算机)又会如何呢? 练习题 19.27 要求为此编写程序。

19.8 编写程序, 利用迭代器演示函数 `rbegin` 和 `rend`。

19.9 编写程序, 实现函数 `data` 和 `c_str`。

19.10 编写程序, 读入几个字符串并只打印以“r”或“ay”结尾的字符串。只考虑小写字母。

19.11 编写程序, 演示按值和按引用传递字符串。

19.12 编写程序, 分别输入姓与名, 然后将它们连接成一个新字符串。

19.13 编写程序, 玩“hangman(吊小人)游戏”(这是一种益智的猜字游戏, 编者注)。程序应该选择一个单词(直接写入程序或从文本文件输入)并显示

```
Guess the word: XXXXXX
```

每一个 X 都代表一个字母。如果用户猜对了, 就显示

```
Congratulations!!! You guessed my word. Play again? yes/no
```

用户可以输入 yes 或 no。如果猜错了, 则显示小人身体的相应部位。

如果有 7 次猜错, 屏幕上就会出现如下所示的结果

```
  O
 /|\
  |
 / \
```

每猜一次都要显示用户所猜的内容。

19.14 编写程序, 输入一个字符串并逆向打印它。将所有大写字母转换为小写, 小写字母转换为大写。

19.15 编写程序, 利用本章介绍的比较函数对一些动物名按字母顺序升序排序。比较中只使用大写字母。

19.16 编写程序, 创建一个字符串的密文。密文为每个字母都要替换为另一个字母替换的消息或单词。例如, 字符串

```
The birds name was sqawk
```

可能变为

```
xms kbypo zhqs fho obrhfu
```

注意空格没有加密。在这个特殊的例子中, 'T' 替换为 'x', 'a' 替换为 'h', 如此等等。大写、小写字母的处理方式相同。使用练习 19.7 中的技巧。

19.17 修改上一个练习题, 允许用户输入两个字符破解密文。第一个字符指定密文中的字母。第二个字母指定用户猜测的字符。例如, 如果用户输入 r g, 那就表示用户 r 猜测为 g。

19.18 编写程序, 输入一个句子并计算句中回文单词的个数。回文是指顺读和倒读都一样的单词。例如, "tree" 不是回文单词, 但 "noon" 是。

19.19 编写程序, 计算一个句子中元音字母的个数。输出每个元音字母的出现频率。

19.20 编写程序, 在一个字符串中间插入字符“\*\*\*\*\*”。

19.21 编写程序, 从一个字符串中删除字符序列“by”和“By”。

19.22 编写程序, 输入一行文本, 用空格替换所有标点符号。然后使用 C 字符串库函数

strok,将这个字符串分解为单个单词。

19.23 编写程序,输入一行文本,并逆向打印文本。在程序中使用迭代器。

19.24 用递归法编写练习题 19.23 中的程序。

19.25 编写程序,演示将迭代器作为参数的函数 erase。

19.26 编写程序,从字符串“abcdefghijklmnopqrstuvwxyz|”中产生如下形状:

```

a
bcb
cdedc
defgfed
efghihgfe
fghijkjihgf
ghijklmlkjihg
hijklmnonmlkjih
ijklmnopqponmlkji
jklmnopqrsrqponmlkj
klmnopqrstutsrqponmlk
lmnopqrstuvwutsrqponml
mnopqrstuvwxyzxwvutsrqponm
nopqrstuvwxyz|zyxwvutsrqpon

```

19.27 在练习题 19.7 中,我们要求大家编写一个简单的加密算法。现在编写程序,用简单的频率替换解密一条“rot13”消息(假定事先不知道密钥)。在加密的短语中出现频率最高的字母应该替换为最常用的英文字母(如 a,e,i,o,u,s,t,r 等等)。并将各种可能性写入文件。怎样才能简化破解过程?如何改进加密机制?

19.28 编写冒泡排序程序用于排序字符串。并在程序中使用函数 swap。

## 第 20 章 标准模板库(STL)

### 学习目标

- 能用标准模板库中的模板容器、容器适配器和“近似容器”
- 能用几十种标准模板库算法编程
- 理解算法如何使用迭代器访问标准模板库容器中的元素
- 熟悉因特网与万维网上的标准模板库资源

### 20.1 标准模板库 STL 简介

除了易于维护和理解外,面向对象的最大好处就是重用、重用、再重用。C++ 标准中包括了包含许多可重用组件的标准库。本章,我们将介绍标准模板库(STL)。我们会讨论标准模板库中三个关键组件:容器(即通用的模板化数据结构)、迭代器和算法。

标准模板库(STL)是惠普公司的 Alexander Stepanov 与 Meng Lee 开发的,是二人在编程领域内研究结果的最好展示,David Musser 在这方面也作出了重要的贡献。标准模板库有望得到广泛使用,因为它已经成为C++ 草案标准中的一部分。

标准模板库涉及的内容较为广泛。我们面临的挑战如何让读者在读完本章后能够开始有效地使用标准模板库。在超过 30 个“活代码”示例程序中,我们涉及了标准模板库中大部分功能。读者将会看到执行中的标准模板库“非常有效”。

第 15 章介绍了数据结构。数据结构是数据的容器(或集合)。在面向对象领域里,数据结构是包含对象的对象。第 8 章捉到的 Array 类包含了基本数据类型 int 的对象。第 12 章介绍了模板后,我们将 Array 类模板化为 `Array<T>`,在此模板中,可以得到无数个 Array 类的实例,如 `Array<int>`,`Array<char>`,`Array<double>`,甚至非基本数据类型的类对象,如 `Array<Employee>`,`Array<SpaceCreature>`,`Array<InventoryItem>`。

对于其他数据结构,如列表、堆栈、队列、优先级队列,如果我们要开发它们的类模板,我们所写的基于这些类模板的程序是否能够很容易地与其他程序员用他们自己的类模板编写的程序相互作用?可能很难。这时就有必要一个标准的模板对象容器库,也就是标准模板库 STL 开发。使用标准模板库可以节省大量时间与精力,产生高质量的程序,这完全归功于“重用世界”。

**性能提示 20.1** 对任何特定的应用来说,都有几种标准模板库容器可供选择。选择最合适的容器以达到最高的性能(即速度与长度的平衡)。在标准模板库设计中,有效性是要考虑的关键因素。

**性能提示 20.2** 标准库提供的功能可以使我们对大量的应用进行有效的操作。但对于特殊性能要求的应用程序,可能需要自行编写自定义的实现方法。

**软件工程知识 20.1** 标准模板库方法允许编写通用程序以使代码不依赖于底层的容器。这种编程方式称为“泛型编程”(generic programming)。

20 世纪 70 年代,我们使用的组件是控制结构与函数。80 年代,我们使用主要使用的类主要来自于各种平台的相关类库。在使用标准模板库的 90 年代后期,我们看到下一个级别的组件,即“独立于平台的类库”。未来 10 年里,我们有希望看到这些类的数量呈指数级增长。

标准模板库是 C++ 标准中的重要组成部分。因此各主要 C++ 编译器提供商都将实现标准模板库。标准模板库一定会得到广泛使用。

在 C 与“原始 C++”里,我们用指针访问数组中的元素。在 C++ 的标准模板库里,我们用迭代器(有些像指针,但比指针更“智能化”)对象访问容器中的元素。迭代器类通用于所有容器。

容器封装了一些基本的操作,但是标准模板库的实现算法独立于容器的。

标准模板库不再使用 new 与 delete,而是用可进行存储分配与释放的分配器。程序员自己可以提供分配器,指定容器如何处理存储管理,但标准模板库提供的默认分配器能满足大多数应用程序的需求。自定义分配器是比较高深的课题,不在本书讨论范围内。

本章将介绍标准模板库。虽然并不完整也不够广泛,但比较友好而且易于理解,能够让你认识到标准模板库的价值,并鼓励你深入学习标准模板库。同样地我们使用了全书使用的“活代码方法”。从理解重用、重用、再重用的角度来说,本章可能是本书中最重要的一章,同时也是篇幅最长的一章。标准模板库容器包含了最常用且最有价值的数据结构。它们都被“模板化”,因此可用它们来容纳与特定应用程序相关的数据类型信息。

第 15 章介绍了数据结构。同时也构建了链表、队列、堆栈和树。我们用指针小心地将连接对象串连在一起。基于指针的代码比较复杂,即使最微小的疏忽也可能导致严重的非法内存访问违例和内存泄露错误,此时编译器是无法检查出来的。实现其他数据结构,如队列、优先级队列、集合、映射等,工作量也较大。

**软件工程知识 20.2** 不必事事从头做起,要用 C++ 标准中的可重用组件来编程。标准模板库将许多最常用的数据结构作为容器,并提供了很多算法,以便程序能够利用这些算法访问这些容器中的数据。

**测试和调试提示 20.1** 编写基于指针的数据结构与算法时,必须自己调试与测试,以确保数据结构、类与算法工作正常。在这种低层次上操纵指针很容易出错。这类代码中,内存泄露与非法内存访问的错误最为常见。对大多数程序员和应用程序来讲,使用标准模板库中预先打包好的、模板化了的数据结构会很有效。使用标准模板库中的代码可节省大量的测试和调试时间。但对大型工程来说,模板的编译时间可能较长。

每种标准模板库容器都有其相关联的成员函数。有些函数适用于所有的标准模板库容器。但其他函数则只适用于特定的容器。我们用类模板 vector, list 与 deque 演示了大部分常见函数。对其他标准模板库容器,我们在示例中介绍了容器的函数。

我们广泛查找了因特网与万维网的资源,并将它们列举在本章最后。

## 20.1.1 容器简介

标准模板库容器的几种类型如图 20.1 所示。这些容器主要分为三类:序列容器、关联容器与容器适配器。序列容器有时也称为“顺序容器”,序列容器较为常用。序列容器与关联容器统称为“第一类容器”。还有 4 种被认为是“近似容器”的容器类型:类似 C 的数组(见第 4 章)、字符串(见第 19 章)、用于维持 1/0 特征值的位集以及用于执行高速数学向量运算的 `valarray`(已经优化,提高了计算性能,但不如“第一类容器”灵活)。这 4 种容器类型被认为是“近似容器”是因为它们与“第一类容器”有相似的功能,但不支持“第一类容器”中所有的功能。

| 标准库容器类         | 说明                      |
|----------------|-------------------------|
| 序列容器           |                         |
| vector         | 从后面快速插入和删除              |
| deque          | 从前面和后面快速插入和删除,直接访问任何元素  |
| list           | 双向链表,在任何地方都可快速插入与删除     |
| 关联容器           |                         |
| set            | 可快速查找,不允许重复值            |
| multiset       | 可快速查找,允许重复值             |
| map            | * 一对一映射,基于关键字快速查找不允许重复值 |
| multimap       | 一对多映射,基于关键字快速查找允许重复值    |
| 容器适配器          |                         |
| stack          | 后进先出(LIFO)              |
| queue          | 先进先出(FIFO)              |
| priority_queue | 优先级最高的元素最先出列            |

图 20.1 标准库容器类

标准模板库经过精心设计使容器能够提供相似功能。有很多通用的操作,如函数 `size`,便适用于所有容器,其他一些操作却只适用于相似容器的子集。此时,可用新类扩充标准模板库。所有标准库容器共有的函数如图 20.2 所示。注意,重载操作符函数 `operator <`, `operator <=`, `operator >`, `operator >=`, `operator ==` 和 `operator !=` 不适用于 `priority_queue`。

| 所有标准模板库容器共有的成员函数 | 说明                                          |
|------------------|---------------------------------------------|
| 默认构造函数           | 对容器进行默认初始化的构造函数。通常,每种容器都有几个构造函数提供初始化容器的不同方法 |
| 复制构造函数           | 将容器初始化为具有相同类型的已有容器的副本                       |
| 析构函数             | 不再需要容器时,为析构函数执行清理工作                         |
| empty            | 容器内没有元素时返回 true,否则返回 false                  |
| max_size         | 返回容器中元素的最大个数                                |
| size             | 返回容器当前所有的元素个数                               |
| operator =       | 将一个容器赋给另一个容器                                |
| operator <       | 如果第一个容器比第二个小,返回 true;否则返回 false             |
| operator <=      | 如果第一个容器小于或等于第二个,返回 true;否则返回 false          |
| operator >       | 如果第一个容器比第二个大,返回 true;否则返回 false             |

(续表)

| 所有标准模板库容器共有的成员函数            | 说明                                                     |
|-----------------------------|--------------------------------------------------------|
| <code>operator &gt;=</code> | 如果第一个容器大于或等于第二个,返回 true;否则返回 false                     |
| <code>operator ==</code>    | 如果第一个容器等于第二个容器,返回 true;否则返回 false                      |
| <code>operator !=</code>    | 如果第一个容器不等于第二个容器,返回 true;否则返回 false                     |
| <code>swap</code>           | 交换两个容器中的元素                                             |
| 下列函数只适用于第一类容器               |                                                        |
| <code>begin</code>          | 此函数的两个版本返回指向容器中第一个元素的 iterator 或 const_iterator        |
| <code>end</code>            | 此函数的两个版本返回指向容器中最后一个元素之后那个位置的 iterator 或 const_iterator |
| <code>rbegin</code>         | 此函数的两个版本返回指向容器中最后一个元素的 iterator 或 const_iterator       |
| <code>rend</code>           | 此函数的两个版本返回指向容器中第一个元素之前那个位置的 iterator 或 const_iterator  |
| <code>erase</code>          | 从容器中删除一个或多个元素                                          |
| <code>clear</code>          | 删除容器中所有元素                                              |

图 20.2 所有标准模板库容器都共有的函数

图 20.3 列出了各标准库容器对应的头文件。这些头文件的内容全部包含在名称空间 `std`(`namespace std`)中。注意,有些 C++ 编译器不支持新式的头文件。这些编译器中,大多数会提供了自己的头文件名。关于编译器对标准模板库的支持信息,参见编译器文档。

**性能提示 20.3** 标准模板库一般避免使用继承与虚拟函数,而用模板来进行常规化编程,以获得更好的执行性能。

| 标准库容器头文件                    |                    |
|-----------------------------|--------------------|
| <code>&lt;vector&gt;</code> |                    |
| <code>&lt;list&gt;</code>   |                    |
| <code>&lt;deque&gt;</code>  |                    |
| <code>&lt;queue&gt;</code>  | 包含了队列与优先级队列        |
| <code>&lt;stack&gt;</code>  |                    |
| <code>&lt;map&gt;</code>    | 包含了映射与复映射          |
| <code>&lt;set&gt;</code>    | 包含了 set 与 multiset |
| <code>&lt;bitset&gt;</code> |                    |

图 20.3 标准库容器头文件

图 20.4 显示了第一类容器中常见的 typedef(用于为长度类型名称创建同义名或别名)。这些 typedef 也用于变量、函数参数和函数返回值的一般性声明。例如,每一种容器中的 `value_type` 始终代表容器中存储的值类型。

| typedef                      | 说明                                         |
|------------------------------|--------------------------------------------|
| <code>value_type</code>      | 存储在容器中元素的类型                                |
| <code>reference</code>       | 对存储在容器中元素的类型的引用                            |
| <code>const_reference</code> | 对存储在容器中元素的类型的常量引用。这种引用只能用于读取容器中的元素以及执行常量操作 |

(续表)

| typedef                | 说明                                           |
|------------------------|----------------------------------------------|
| pointer                | 指向存储在容器中元素类型的指针                              |
| iterator               | 指向存储在容器中元素类型的迭代器                             |
| const_iterator         | 指向存储在容器中元素类型的常量迭代器,只能用于读取元素                  |
| reverse_iterator       | 指向存储在容器中元素类型的反向迭代器。这种迭代器用于反向迭代               |
| const_reverse_iterator | 指向存储在容器中元素类型的常量反向迭代器,只能用来读取元素。这种迭代器用于反向遍历容器  |
| difference_type        | 指向同一容器的两个迭代器之差值的类型(list 与关联容器没有定义 operator-) |
| size_type              | 用于计算容器中的条目个数以及为序列容器编排索引的类型(不能对 list 编排索引)    |

图 20.4 第一类容器中常见的 typedef

**可移植性提示 20.1** 标准模板库必然会成为容器编程的优选方法。使用标准模板库编程可增强代码的可移植性。

**性能提示 20.4** 了解标准模板库组件。针对特定问题,选择最合适的容器,可提高性能和减少内存需求。

打算使用标准模板库容器时,要确保容器中存储的元素类型支持最基本的功能集合。将一个元素插入容器时,便生成了该元素的副本。因此,这个元素类型应该提供它自己的复制构造函数与赋值操作。注意:只有默认的成员复制函数不对这种元素类型执行正确的复制操作时,才有必要如此。此外,关联容器和许多算法都需要对比较元素,因此元素类型也应提供等于操作符(=)与小于操作符(<)。

**软件工程知识 20.3** 从技术上讲,容器中存储的元素并不需要等于(=)和小于(>)操作符,除非需要比较元素。然而,用模板创建代码时,有些编译器需要模板的所有部分都要定义,有些编译器则只需要在程序中定义实际使用的部分模板。

## 20.1.2 迭代器简介

迭代器与指针有许多共性,它用于指向第一类容器中的元素(还有其他用途)。迭代器保存它所操作的特定容器所需的状态信息,因此实现迭代器与每种容器相适应的。然而,有些迭代器操作在所有容器中都一样。例如,间接引用操作符(\*)间接引用迭代器以便能使用迭代器指向的那个元素。在迭代器上使用操作符++可将迭代器移到容器中的下一个元素(如同在数组中递增指针使其指向数组中下一个元素)。

标准模板库第一类容器提供了成员函数 begin()与 end()。函数 begin()返回指向容器中第一个元素的迭代器。函数 end()返回指向容器最后一个元素之后那个元素(此元素不存在)的迭代器。如果迭代器 i 指向某一元素,那么 ++i 将指向下一个元素。\*i 为 i 所指向的元素。函数 end()返回的迭代器只能用于是否相等的比较中确定“移动”的迭代器(在这里为 i)是否到达了容器尾。

我们用 iterator 类型的对象指向一个能修改的容器元素。同时也用 const\_iterator 类型的对象指向一个不能修改的容器元素。

我们在“序列”(也称为“排列”)中使用迭代器。这些序列可能在容器中,也可能是输入

序列或输出序列。图 20.5 中的程序演示了如何用 `istream_iterator` 从标准输入获取输入(输入到程序中的一列数据)和如何用 `ostream_iterator` 将输出(程序中用来输出的一列数据)送往标准输出设备。程序让用户从键盘输入两个整数并显示它们的和。(注意:本章示例中标准模板库函数的使用与标准模板库容器对象的定义之前,我们使用了“`std::`”前缀,而不是像前面的程序那样,将 `using` 语句放在程序的开头。由于编译器之间的差别以及标准模板库产生的代码较为复杂,所以很难构建适当的 `using` 语句集合以保证程序编译时没有错误。为使这些程序能更多的平台上编译,我们选择了“`std::`”前缀法。)

```

1 //Fig. 20.5: fig20_05.cpp
2 //Demonstrating input and output with iterators.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iterator>
10
11 int main()
12 {
13     cout << "Enter two integers: ";
14
15     std::istream_iterator< int > inputInt( cin );
16     int number1, number2;
17
18     number1 = *inputInt; //read first int from standard input
19     ++inputInt;          //move iterator to next input value
20     number2 = *inputInt; //read next int from standard input
21
22     cout << "The sum is: ";
23
24     std::ostream_iterator< int > outputInt( cout );
25
26     *outputInt = number1 + number2; //output result to cout
27     cout << endl;
28     return 0;
29 }
```

输出结果:

```

Enter two integers: 12 25
The sum is: 37
```

图 20.5 输入与输出流迭代器示例

第 15 行

```
std::istream_iterator< int > inputInt( cin );
```

创建了迭代器 `istream_iterator`, 该迭代器能从标准输入对象 `cin` 中以类型安全的方式获取(输入)整型数据。第 18 行

```
number1 = *inputInt; //read first int from standard input
```



间接引用迭代器 `inputInt` 来从 `cin` 中读取第一个整数并将它赋给 `number1`。注意使用间接引用操作符 `*` 从与 `inputInt` 相联的数据流中获得数值。这与间接引用指针相似。第 19 行

```
* ++inputInt; //move iterator to next input value
```

把迭代器 `inputInt` 移到输入流的下一个值。第 20 行

```
number2 = *inputInt; //read next int from standard input
```

从 `inputInt` 输入下一个整数并将赋给 `number2`。

第 24 行

```
std::ostream_iterator< int > outputInt( cout );
```

创建了迭代器 `ostream_iterator`, 该迭代器能将整型值插入(输出)到标准输出对象 `cout`。

第 26 行

```
*outputInt = number1 + number2; //output result
```

将 `number1` 与 `number2` 的和赋给 `*output`, 从而把一个整数输出到 `cout`。注意在赋值语句中用间接引用操作符 `*` 将 `*outputInt` 作为左值。如果想用 `outputInt` 输出另一个整数, 就必须用 `++` 递增迭代器(前置自增和后置自增均可)。

**测试和调试提示 20.2** 常量迭代器的间接引用操作符(`*`)返回对容器元素的常量引用, 因此不允许非常量成员函数使用它。

**常见编程错误 20.1** 试图间接引用容器之外的迭代器会导致运行时逻辑错误。尤其不能间接引用或自增 `end()` 返回的迭代器。

**常见编程错误 20.2** 试图为常量容器创建非常量迭代器是语法错误。

图 20.6 列出了标准模板库所用的迭代器类别。每一类迭代器都有特定的一系列功能。

| 类别      | 说明                                                                              |
|---------|---------------------------------------------------------------------------------|
| 输入迭代器   | 用于从容器中读取元素。输入迭代器只能一次一个元素地向前移动(即从容器头移到容器尾)。它只支持“单次遍历”算法(即同一个输入迭代器不能用于两次遍历一个序列容器) |
| 输出迭代器   | 用于将元素写入到容器中。输出迭代器只能一次一个元素地向前移动。它仅支持“单次遍历”算法(即同一个输出迭代器不能用于两次遍历一个序列容器)            |
| 正向迭代器   | 具有输入迭代器与输出迭代器两者的功能,并能保持它们在容器中的位置(如状态信息)                                         |
| 双向迭代器   | 除了具有正向迭代器的功能外,还能向后移动(即从容器尾移动到容器头)。它支持“多次遍历”算法                                   |
| 随机访问迭代器 | 除了具有双向迭代器的功能,还能直接访问容器中的任意元素,即可以任意向前或向后跳转                                        |

图 20.6 迭代器类别

迭代器类别层次结构如图 20.7 所示。顺着其结构从上往下看,每一类迭代器都支持其上一层迭代器的功能。因此“功能最弱”的迭代器位于顶部,功能最强大的迭代器则位于底部。注意图 20.7 并非继承层次结构。

各容器支持的迭代器类别决定了容器是否能使用标准模板库的特定算法。支持随机访问的迭代器可用标准模板库中所有的算法。可以看到,在大多数算法(包括那些需要随机访问迭代器的算法)中数组指针都可以取代迭代器。每个标准模板库容器支持的迭代器类别如图 20.8 所示。注意只有 `vector`, `deque`, `list`, `set`, `multiset`, `map` 和 `multimap`(即第一类容器)能够用迭代器遍历。

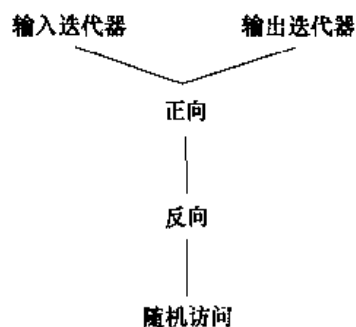


图 20.7 迭代器类别结构图

| 容器             | 支持的迭代器类型 |
|----------------|----------|
| 序列容器           |          |
| vector         | 随机访问迭代器  |
| deque          | 随机访问迭代器  |
| list           | 双向迭代器    |
| 关联容器           |          |
| set            | 双向迭代器    |
| multiset       | 双向迭代器    |
| map            | 双向迭代器    |
| multimap       | 双向迭代器    |
| 容器适配器          |          |
| stack          | 不支持迭代器   |
| queue          | 不支持迭代器   |
| priority_queue | 不支持迭代器   |

图 20.8 各种标准库容器支持的迭代器类型

**软件工程知识 20.4** 使用能满足基本性能需求的“功能最弱”的迭代器有利于产生最具重用性的组件。

标准模板库容器的类定义中的预定义迭代器 typedef 如图 20.9 所示。但并非每种容器都定义了这些 typedef。我们用迭代器的常量版本遍历只读容器,以及使用反向迭代器反向遍历容器。

| 预定义迭代器 typedef         | ++ 方向 | 功能  |
|------------------------|-------|-----|
| iterator               | 向前    | 读/写 |
| const_iterator         | 向前    | 读   |
| reverse_iterator       | 向后    | 读/写 |
| const_reverse_iterator | 向后    | 读   |

图 20.9 预定义迭代器 typedef

**测试和调试提示 20.3** 用 const\_iterator 进行的操作返回常量引用以防止修改所操作的容器元素。能用 const\_iterator 时,尽量不用 iterator。这是最低权限原则的又一个实例。

各类迭代器的操作如图 20.10 所示。注意,图中各类迭代器的操作都包含其前一个迭代器的所有操作。此外,输入迭代器与输出迭代器不能保存迭代器,并使用保存值。

| 迭代器操作                   | 说明                                                                                                                                              |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 所有迭代器                   |                                                                                                                                                 |
| <code>++p</code>        | 前置自增迭代器                                                                                                                                         |
| <code>p++</code>        | 后置自增迭代器                                                                                                                                         |
| 输入迭代器                   |                                                                                                                                                 |
| <code>*p</code>         | 间接引用迭代器作为左值                                                                                                                                     |
| <code>p = p1</code>     | 将一个迭代器赋给另一个                                                                                                                                     |
| <code>p == p1</code>    | 比较两个迭代器是否相等                                                                                                                                     |
| <code>p != p1</code>    | 比较两个迭代器是否不相等                                                                                                                                    |
| 输出迭代器                   |                                                                                                                                                 |
| <code>*p</code>         | 间接引用迭代器(作为左值)                                                                                                                                   |
| <code>p = p1</code>     | 将一个迭代器赋给另一个                                                                                                                                     |
| 正向迭代器                   |                                                                                                                                                 |
| 双向迭代器                   |                                                                                                                                                 |
| <code>--p</code>        | 前置自减迭代器                                                                                                                                         |
| <code>p--</code>        | 后置自减迭代器                                                                                                                                         |
| 随机访问迭代器                 |                                                                                                                                                 |
| <code>p += i</code>     | 将迭代器 <code>p</code> 的位置递增 <code>i</code>                                                                                                        |
| <code>p -= i</code>     | 将迭代器 <code>p</code> 的位置递减位 <code>i</code>                                                                                                       |
| <code>p + i</code>      | 在 <code>p</code> 位自增 <code>i</code> 位之后的迭代器                                                                                                     |
| <code>p - i</code>      | 在 <code>p</code> 位自减 <code>i</code> 位之后的迭代器                                                                                                     |
| <code>p[i]</code>       | 返回基点为 <code>p</code> , 偏移 <code>i</code> 位置的元素                                                                                                  |
| <code>p &lt; p1</code>  | 如果迭代器 <code>p</code> 小于迭代器 <code>p1</code> (即在容器中 <code>p</code> 在 <code>p1</code> 之前)则返回 <code>true</code> ; 否则返回 <code>false</code>           |
| <code>p &lt;= p1</code> | 如果迭代器 <code>p</code> 小于或等于迭代器 <code>p1</code> (即在容器中 <code>p</code> 在 <code>p1</code> 之前或处于同一位置)则返回 <code>true</code> ; 否则返回 <code>false</code> |
| <code>p &gt; p1</code>  | 如果迭代器 <code>p</code> 大于迭代器 <code>p1</code> (即在容器中 <code>p</code> 在 <code>p1</code> 之后)则返回 <code>true</code> ; 否则返回 <code>false</code>           |
| <code>p &gt;= p1</code> | 如果迭代器 <code>p</code> 大于或等于迭代器 <code>p1</code> (即在容器中 <code>p</code> 在 <code>p1</code> 之后或处于同一位置)则返回 <code>true</code> ; 否则返回 <code>false</code> |

图 20.10 各类迭代器能执行的操作

### 20.1.3 算法简介

标准模板库一个重要的方面就是它提供了适用于多种容器的通用算法,它提供了许多常用于操纵容器的算法。如插入、删除、查找、排序以及其他适用于部分或全部标准模板库容器的操作。

标准模板库包括了近 70 种算法。我们提供了大多数算法的“活代码”例子并总结了其余的算法,算法只能通过迭代器间接操纵容器中的元素。很多算法用于操作称为迭代器对中的元素序列。所谓迭代器对,是指第一个迭代器指向序列中的第一个元素,第二个指向序列中最后一个元素之后的元素。此外,也可以自己创建相似的算法以便将其用于标准模板库容器和迭代器。

容器成员函数 `begin()` 返回指向容器中第一个元素的迭代器,成员函数 `end()` 返回容器中最后一个元素之后那个位置的迭代器。算法通常返回迭代器。

例如,算法 `find()` 查找一个元素并返回指向这个元素的迭代器。找不到 `find()` 就返回

end() 迭代器,这用于确定元素是否已找到(返回 end() 意味着已查找了整个容器)。算法 find() 适用于所有标准模板库容器。

**软件工程知识 20.5** 标准模板库的实现很简洁。直到现在,类设计者还是通过编写容器的算法成员函数联系算法与容器。但标准模板库采取了另外一种方式。在标准模板库中,算法与容器是分离的,算法通过迭代器间接操纵容器中的元素。这种分离简化了编写适用于多种容器类的算法。

标准模板库算法创造了另一种有利于重用的环境,使用拥有大量常见算法的这个算法集可以为程序员节省大量时间与精力。

如果一种算法使用功能稍弱的迭代器,那就可以用于支持功能更强大的迭代器容器中。有些算法需要功能强大的迭代器,如算法 sort 就需要随机访问迭代器。

**软件工程知识 20.6** 标准模板库具有可扩展性。无须任何更改,即可将算法直接加入标准模板库标准模板库容器。

**软件工程知识 20.7** 标准模板库算法能够操作容器以及基于指针的 C 风格数组。

**可移植性提示 20.2** 因为标准模板库算法通过迭代器间接操作容器,所以一种算法可用于不同容器。

图 20.11 列出了许多“改变序列”算法(即算法会修改它所操作的容器)。

| “改变序列”算法        |                   |                    |
|-----------------|-------------------|--------------------|
| copy()          | remove()          | reverse_copy()     |
| copy_backward() | remove_copy()     | rotate()           |
| fill()          | remove_copy_if()  | rotate_copy()      |
| fill_n()        | remove_if()       | stable_partition() |
| generate()      | replace()         | swap()             |
| generate_n()    | replace_copy()    | swap_ranges()      |
| iter_swap()     | replace_copy_if() | transform()        |
| partition()     | replace_if()      | unique()           |
| random_shuffle  | reverse()         | unique_copy()      |

图 20.11 “改变序列”算法

图 20.12 列出了许多“不改变序列”算法(即算法不会修改它所操作的容器)。

| “不改变序列”算法       |                 |            |
|-----------------|-----------------|------------|
| adjacent_find() | find()          | find_if()  |
| count()         | find_each()     | mismatch() |
| count_if()      | find_end()      | search()   |
| equal()         | find_first_of() | search_n() |

图 20.12 “不改变序列”算法

图 20.13 列出了头文件 <numeric> 中的数字算法。

---

头文件 <numeric> 中的数字算法

---

```
accumulate()  
inner_product()  
partial_sum()  
adjacent_difference()
```

---

图 20.13 头文件 <numeric> 中的数字算法

## 20.2 序列容器

C++ 标准模板库提供了 3 种序列容器:vector, list 和 deque。vector 类与 deque 类都基于数组, list 类与第 15 章描述的链表数据结构类似, 但比链表更健壮。

标准模板库中最常用的容器之一是 vector。vector 类是我们在第 8 章创建的智能数组的改进版, vector 可以动态改变长度, 与 C 与 C++ “原始”数组(参见第 4 章)不同的是, vector 能够相互赋值。这对基于指针的 C 风格数组来说是不可能的, 因为数组名为常量指针, 因此不能成为赋值对象。与 C 风格数组相同, vector 下标操作也不执行范围检查, 但 vector 类可以通过成员函数 at 提供了这种检查功能。

**性能提示 20.5** 在 vector 结尾执行插入操作是非常高效的。vector 仅在需要时增长以容纳新的元素。在 vector 中间执行插入(或删除)操作代价较高:vector 插入(或删除)点之后所有部分都要进行相应的移动, 因为 vector 元素存放在连续的内存空间。这一点与 C 或 C++ “原始”数组相似。

图 20.2 显示的操作通用于所有标准模板库容器。除这些操作外, 各容器还提供了部分其他功能, 这些功能中大多数可用于多种容器。但这些操作在不同的容器中的执行时效率并不相同, 程序员经常要选择最适合自己的应用程序的容器。

**性能提示 20.6** 需要在容器的两端频繁执行插入和删除操作的应用程序通常使用 deque 而不是 vector。虽然可以在两者的头部与结尾进行插入和删除操作, 但 vector 类在头部执行插入和删除操作比 deque 类更有效率。

**性能提示 20.7** 需要在容器中间和/或两端进行插入和删除操作的应用程序通常使用 list, 因为 list 可在任何地方进行插入或删除操作。

除了图 20.2 介绍的通用操作外, 序列容器还有其他几种通用操作:操作 front 返回对容器中第一个元素的引用, 操作 back 返回对容器中最后一个元素的引用, 操作 push\_back 将一个元素插入容器结尾, pop\_back 删除容器中的最后一个元素。

### 20.2.1 vector 类序列容器

vector 类提供了一种占用连续内存空间的数据结构。这样就可以通过下标操作符[]高效而直接地访问 vector 中的任意元素, 这一点与 C 或 C++ “原始”数组完全一样。容器中数据需要排序并通过下标操作符访问时, 通常使用 vector 类。当 vector 耗尽内存时, 它会分配一块更大的连续内存空间, 并将原来的数据复制到新分配的内存中然后释放原来的内存。

**性能提示 20.8** 使用 vector 容器以达到最好的随机访问性能。

**性能提示 20.9** vector 对象能够使用重载的下标操作符[]实现快速按索引访问,因为它们像 C 或 C++“原始”数组一样,存放在连续的内存空间内。

**性能提示 20.10** 一次插入多个元素比一次插入一个元素更有效。

每种容器最重要的部分是它所支持的迭代器类型。这决定了哪些算法能够用于这种容器。vector 支持随机访问迭代器,即图 20.10 中的所有迭代器操作都能用于 vector 迭代器。所有的标准模板库算法都能操作 vector。vector 迭代器通常实现为指向 vector 中元素的指针。每一种将迭代器作为参数的标准模板库算法都需要这些迭代器提供基本功能集合。例如,如果一种算法需要正向迭代器,那就能操作任何一个能够提供正向迭代器、双向迭代器或随机访问迭代器的容器。只要容器支持算法的最弱迭代器功能,算法就能操作这个容器。

图 20.14 演示了 vector 类模板的几个函数。这些函数也适用于标准库中的第一类容器。要使用 vector 类,必须包含头文件 <vector>。

```

1 //Fig. 20.14: fig20_14.cpp
2 //Testing Standard Library vector class template
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <vector>
10
11 template < class T >
12 void printVector( const std::vector< T > &vec );
13
14 int main()
15 {
16     const int SIZE = 6;
17     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
18     std::vector< int > v;
19
20     cout << "The initial size of v is: " << v.size()
21         << " \nThe initial capacity of v is: " << v.capacity();
22     v.push_back( 2 ); //method push_back() is in
23     v.push_back( 3 ); //every sequence collection
24     v.push_back( 4 );
25     cout << " \nThe size of v is: " << v.size()
26         << " \nThe capacity of v is: " << v.capacity();
27     cout << " \n\nContents of array a using pointer notation: ";
28
29     for ( int *ptr = a; ptr != a + SIZE; ++ptr )
30         cout << *ptr << " ";
31
32     cout << " \nContents of vector v using iterator notation: ";
33     printVector( v );

```

```

34
35     cout << " \nReversed contents of vector v: ";
36
37     std::vector< int >::reverse_iterator p2;
38
39     for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
40         cout << *p2 << " ";
41
42     cout << endl;
43     return 0;
44 }
45
46 template < class T >
47 void printVector( const std::vector< T > &vec )
48 {
49     std::vector< T >::const_iterator p1;
50
51     for ( p1 = vec.begin(); p1 != vec.end(); p1 ++ )
52         cout << *p1 << " ";
53 }
54
55 /*
56     Demonstrates
57         Vector declaration
58         passing vector to method via const reference
59         push_back
60         size
61         capacity
62         begin
63         end
64         rbegin
65         rend
66 */

```

输出结果:

```

The initial size of v is: 0
The initial capacity of v is : 0
The size of v is: 3
The capacity of v is: 4

```

```

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2

```

图 20.14 vector 类模板的函数示例

### 第 18 行

```
std::vector< int > v;
```

定义了 `vector` 类的对象实例 `v`, 用于存储整型值。实例化该对象时, 创建了一个空的 `vector`, 该 `vector` 长度为 0 (即存放在 `vector` 中的元素个数)、容量为 0 (不为 `vector` 分配更多内存的前提下, `vector` 所能存储的元素个数)。

## 第 20 行和第 21 行

```
cout << "The initial size of v is: " << v.size()
    << " \nThe initial capacity of v is: " << v.capacity();
```

演示了函数 `size` 与 `capacity`, 这两个函数起初都返回 0。函数 `size` (每种容器都有该函数) 返回容器中当前存放的元素个数, 函数 `capacity` 返回动态改变 `vector` 长度以容纳更多元素之前, 该 `vector` 中能存放的元素个数。

## 第 22 ~ 24 行

```
v.push_back(2); //every sequence collection method push_back() is in
v.push_back(3);
v.push_back(4);
```

用函数 `push_back` (每种序列容器都有该函数) 将一个元素添加到 `vector` 末尾。如果在已满的向量加入一个元素, 向量的长度会增长 (某些标准模板库会自动将 `vector` 的长度增加一倍)。

**性能提示 20.11** 需要更多空间时, 自动将 `vector` 长度增加一倍, 可能会浪费空间。例如, 加入一个新元素时, 一个拥有 1 000 000 个元素的已满向量会将其长度增大到足以容纳 2 000 000 个元素的长度。这会留下 999 999 个元素的空闲空间。程序员可使用 `resize()` 更好地控制空间的分配。

第 25 行和第 26 行用函数 `size` 与 `capacity` 显示 `push_back` 操作之后, `vector` 新的长度与容量。函数 `size` 返回 3 (即增加到向量中的元素的个数)。函数 `capacity` 返回 4, 表示不为向量分配更多空间时可增加一个元素。增加第一个元素时, `v` 的长度、容量均为 1。增加第二个元素时, `v` 的长度与容量变为 2。增加第三个元素时, `v` 的长度变为 3, 容量变为 4。如果我们再增加两个元素, `v` 的长度将变为 5, 容量变为 8。每次当向量已满而继续向其中增加一个元素时, 向量的容量就会自动翻倍。

第 29 行和第 30 行演示了如何用指针和指针算法输出数组中的内容。第 33 行调用函数 `printVector` 通过迭代器输出 `vector` 中的内容。函数模板 `printVector` 的定义从第 46 行开始。函数取对 `vector` 的常量引用作为参数。第 49 行

```
std::vector<T>::const_iterator pl;
```

定义了常量迭代器 `pl` 遍历向量并输出向量内容。`const_iterator` 使程序可以读取但不能修改 `vector` 元素。第 51 行和第 52 行的 `for` 循环结构

```
for (pl = vec.begin(); pl != vec.end(); pl++)
    cout << *pl << " ";
```

初始化为 `pl`, 这是利用 `vector` 成员函数 `begin` 来实现的。该函数将一个 `const_iterator` 返回 `vector` 中的第一个元素 (`begin` 还有另一个版本, 返回可用于非常量容器的迭代器)。只要 `pl` 不超过 `vector` 尾, 循环就会继续。这可以通过比较 `pl` 与函数 `vec.end()` 的结果 (指向 `vector` 中最后一个元素之后那个位置的常量迭代器) 来确定 (函数 `end` 与 `begin` 一样, `end` 的另一版本返回非常量迭代器)。如果 `pl` 等于 `vec.end()` 返回值, 就表明已经到了 `vector` 尾。所有第一类容器都有函数 `begin` 和 `end`。循环体间接引用 `pl` 获得容器中当前元素的值。表达式



pl ++ 将迭代器移到 vector 的下一个元素。

**测试和调试提示 20.4** 只有随机访问迭代器支持小于操作符(<),所以最好用!=与 end()测试是否到达容器尾。

第 37 行

```
std::vector< int >::reverse_iterator p2;
```

声明了能够反向遍历向量的反向迭代器 reverse\_iterator,所有第一类容器都支持这种迭代器。

第 39 行和第 40 行

```
for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
    cout << *p2 << " ";
```

用类似于函数 printVector 的 for 循环结构遍历向量。在此循环中,函数 rbegin()与 rend()表示进行反向输出的元素的范围。与函数 begin 和 end 一样,rbegin 和 rend 能够根据容器是否为常量而返回常量反向迭代器 const\_reverse\_iterator 与反向迭代器 reverse\_iterator。

图 20.15 演示了用于读取和操作 vector 元素的函数。第 16 行

```
std::vector< int > v(a, a + SIZE);
```

使用重载的 vector 构造函数,该函数带有两个迭代器参数。记住,数组的指针可以用作迭代器。这条语句创建了整型 vector v,并用位置 a 到位置 a + SIZE(不包括位置 a + SIZE)的整型数组 a 对它进行了初始化。

```
1 //Fig.20.15: fig20_15.cpp
2 //Testing Standard Library vector class template
3 //element-manipulation functions
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <vector>
10 #include <algorithm>
11
12 int main()
13 {
14     const int SIZE = 6;
15     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
16     std::vector< int > v( a, a + SIZE );
17     std::ostream_iterator< int > output( cout, " " );
18     cout << "Vector v contains: ";
19     std::copy( v.begin(), v.end(), output );
20
21     cout << "\nFirst element of v: " << v.front()
22         << "\nLast element of v: " << v.back();
23
24     v[ 0 ] = 7; //set first element to 7
25     v.at( 2 ) = 10; //set element at position 2 to 10
26     v.insert( v.begin() + 1, 22 ); //insert 22 as 2nd element
```

```

27     cout << " \nContents of vector v after changes: ";
28     std::copy( v.begin(), v.end(), output );
29
30     try {
31         v.at( 100 ) = 777;    //access element out of range
32     }
33     catch ( std::out_of_range e ) {
34         cout << " \nException: " << e.what();
35     }
36
37     v.erase( v.begin() );
38     cout << " \nContents of vector v after erase: ";
39     std::copy( v.begin(), v.end(), output );
40     v.erase( v.begin(), v.end() );
41     cout << " \nAfter erase, vector v "
42         << ( v.empty() ? "is" : "is not" ) << " empty";
43
44     v.insert( v.begin(), a, a + SIZE );
45     cout << " \nContents of vector v before clear: ";
46     std::copy( v.begin(), v.end(), output );
47     v.clear();    //clear calls erase to empty a collection
48     cout << " \nAfter clear, vector v "
49         << ( v.empty() ? "is" : "is not" ) << " empty";
50
51     cout << endl;
52     return 0;
53 }

```

输出结果:

```

Vector v contains: 1 2 3 4 5 6
First element of v: 1
Last element of v: 6
Contents of vector v after changes: 7 22 2 10 4 5 6
Exception: invalid vector<T> subscript
Contents of vector v after erase: 22 2 10 4 5 6
After erase, vector v is empty
Contents of vector v before clear: 1 2 3 4 5 6
After clear, vector v is empty

```

图 20.15 标准库中 vector 类模板元素操作函数示例

第 17 行

```
std::ostream_iterator< int > output(cout, " ");
```

定义了输出流迭代器 output, 该迭代器将通过 cout 输出用单个空格进行分隔的整数。输出流迭代器是类型安全的, 只输出整型或兼容类型的值。构造函数的第一个参数指定了输出流。第二个参数为一个字符串, 用于指定输出值的分隔字符(在这里为一个空格)。本例中, 我们将用输出流迭代器输出 vector 中的内容。

第 19 行

```
std::copy(v.begin(), v.end(), output);
```

用标准库中的算法 copy 将 vector v 的全部内容输出到标准输出设备。算法 copy 将复制容器

中第一个参数中迭代器指定的位置到第二个参数中迭代器指定的位置(不包括这个位置)之间的所有元素。第一个与第二个参数必须满足输入迭代器的要求(即可通过它们读取容器中的元素)。此外,对第一个迭代器进行递增操作 ++ 最终会导致第一个迭代器到达容器中第二个迭代器指向的位置。这些元素被复制到最后一个参数指定的输出迭代器(可通过它存储或输出值)指定的位置。在这里,输出迭代器 output 与 cout 相联,所以元素被复制到标准输出设备。要使用标准库中的算法,必须包含头文件 `<algorithm>`。

第 21 行和第 22 行分别用函数 `front` 与 `back`(所有序列容器都有这两个函数)确定 `vector` 中的第一个与最后一个元素。

**常见编程错误 20.3** `vector` 不能为空,否则函数 `front` 与 `back` 会返回不确定的结果。

第 24 行和第 25 行

```
v[0] = 7;    //set first element to 7
v.at(2) = 10 //set element at position 2 to 10
```

演示了对 `vector` 进行下标操作的两种方法(也适用于 `deque`)。第 24 行使用重载的下标操作符返回对指定位置值的引用或常量引用(与容器是否为常量有关)。函数 `at` 执行相同的操作,但增加了边界检查。它首先检查其参数值并确定参数是否在 `vector` 的边界之内。如果不在,函数 `at` 会抛出 `out_of_range` 异常(参见第 30~35 行)。一些标准模板库的异常如图 20.16 所示(第 13 章讨论了标准模板库的异常类型)。

| 标准模板库异常类型                     | 说明                                                |
|-------------------------------|---------------------------------------------------|
| <code>out_of_range</code>     | 表示下标超出范围,如在向量的函数 <code>at</code> 中指定一个无效的下标       |
| <code>invalid_argument</code> | 表示传递给函数的参数无效                                      |
| <code>length_error</code>     | 表示试图创建过长的容器、字符串等                                  |
| <code>bad_alloc</code>        | 表示试图用 <code>new</code> (或分配器)分配内存时,因为内存不足而导致的操作失败 |

图 20.16 标准模板库异常类型

第 26 行

```
v.insert(v.begin() + 1, 22); //insert 22 as 2nd element
```

用 3 个插入函数中的一个(所有序列容器都有)。这条语句将数值 22 插入第一个参数中迭代器所指定位置的元素之前。本例中,迭代器指向 `vector` 的第二个元素,因此 22 就作为第二个元素插入,原来的第二个元素就变成第 3 个元素。另外两个 `insert` 函数在容器的特定位置插入同一个值的多个副本或将特定位置起的另一个容器(或数组)的一系列值插入原始容器中。

第 37 行和第 40 行

```
v.erase(v.begin());
v.erase(v.begin(), v.end());
```

用了两个 `erase` 函数(所有第一类容器都有)。第 37 行表明删除迭代器参数指定位置上的元素(本例中,这个元素为 `vector` 的开头)。第 40 行表明删除第一个参数指定的位置到第二个参数指定的位置(不包括这个位置)之间的元素。本例中,`vector` 中所有元素都被删除。第 42 行用函数 `empty`(包括适配器在内的所有容器都有该函数)确认了 `vector` 为空。

常见编程错误 20.4 删除其中包含指向动态分配对象指针的元素时,没有删除这个对象。

第 44 行

```
v.insert(v.begin(), a, a + SIZE);
```

使用了函数 `insert`, 该版本的 `insert` 第二个与第 3 个参数指定了要插入 `vector` 一系列数值(可能来自容器, 但本例中的数值来自于数组 `a`) 的起始与终止位置。记住, 终止位置指定了要插入的序列中最后一个元素之后的那个位置。复制延续到终止位置为止(但不包括这个位置)。

第 47 行

```
v.clear(); //clear calls erase to empty a container
```

用函数 `clear`(第一类容器都有该函数)清空容器。它实际上调用了第 40 行的函数 `insert` 执行清空操作。

注意, 还有部分通用于所有容器和序列容量的函数。我们随后将讨论其中的一部分, 同时还将讨论许多用于特定容器的函数。

## 20.2.2 list 顺序容器

`list` 序列容器能在容器的任意位置有效地进行插入与删除操作。如果大部分的插入与删除操作在容器的两端进行, 使用 `deque`(第 20.2.3 节)将更有效。`list` 类实现为双向链表: 即列表中的每个节点包含了指向列表中前一个节点的指针和指向列表中下一个节点的指针。所以 `list` 类支持双向迭代器而允许容器向前或向后遍历。任何一种需要输入、输出或双向迭代器的算法都能操纵 `list`。`list` 的许多成员函数都将容器中的元素作为有序的元素集进行操作。

除图 20.2 列出的适用于所有标准模板库容器的成员函数和第 20.5 节介绍的适用于所有顺序容器的成员函数外, `list` 类还包含了 8 个成员函数: `splice`, `push_front`, `pop_front`, `remove`, `unique`, `merge`, `reverse` 和 `sort`。图 20.17 演示了 `list` 类的一些特性。记住, 图 20.14 与 20.15 描述的大部分函数也可用于 `list` 类。要使用 `list` 类, 必须包含头文件 `<list>`。

```
1 //Fig. 20.17: fig20_17.cpp
2 //Testing Standard Library class list
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <list>
9 #include <algorithm>
10
11 template < class T >
12 void printList( const std::list< T > &listRef );
13
14 int main()
15 {
16     const int SIZE = 4;
17     int a[ SIZE ] = { 2, 6, 4, 8 };
```

```
18  std::list< int > values, otherValues;
19
20  values.push_front( 1 );
21  values.push_front( 2 );
22  values.push_back( 4 );
23  values.push_back( 3 );
24
25  cout << "values contains: ";
26  printList( values );
27  values.sort();
28  cout << "\nvalues after sorting contains: ";
29  printList( values );
30
31  otherValues.insert( otherValues.begin(), a, a + SIZE );
32  cout << "\notherValues contains: ";
33  printList( otherValues );
34  values.splice( values.end(), otherValues );
35  cout << "\nAfter splice values contains: ";
36  printList( values );
37
38  values.sort();
39  cout << "\nvalues contains: ";
40  printList( values );
41  otherValues.insert( otherValues.begin(), a, a + SIZE );
42  otherValues.sort();
43  cout << "\notherValues contains: ";
44  printList( otherValues );
45  values.merge( otherValues );
46  cout << "\nAfter merge:\n  values contains: ";
47  printList( values );
48  cout << "\n  otherValues contains: ";
49  printList( otherValues );
50
51  values.pop_front();
52  values.pop_back(); //all sequence containers
53  cout << "\nAfter pop_front and pop_back values contains:\n";
54  printList( values );
55
56  values.unique();
57  cout << "\nAfter unique values contains: ";
58  printList( values );
59
60  //method swap is available in all containers
61  values.swap( otherValues );
62  cout << "\nAfter swap:\n  values contains: ";
63  printList( values );
64  cout << "\n  otherValues contains: ";
65  printList( otherValues );
66
67  values.assign( otherValues.begin(), otherValues.end() );
68  cout << "\nAfter assign values contains: ";
```

```

69     printList( values );
70
71     values.merge( otherValues );
72     cout << "\nvalues contains: ";
73     printList( values );
74     values.remove( 4 );
75     cout << "\nAfter remove( 4 ) values contains: ";
76     printList( values );
77     cout << endl;
78     return 0;
79 |
80
81 template < class T >
82 void printList( const std::list< T > &listRef )
83 |
84     if ( listRef.empty() )
85         cout << "List is empty";
86     else {
87         std::ostream_iterator< T > output( cout, " " );
88         std::copy( listRef.begin(), listRef.end(), output );
89     |
90 }

```

输出结果:

```

values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
otherValues contains: 2 6 4 8
After splice values contains: 1 2 3 4 2 6 4 8
values contains: 1 2 2 3 4 4 6 8
otherValues contains: 2 4 6 8
After merge;
    values contains: 1 2 2 2 3 4 4 4 6 6 8 8
    otherValues contains: List is empty
After pop_front and pop_back values contains:
2 2 2 3 4 4 4 6 6 8
After unique values contains: 2 3 4 6 8
After swap;
    values contains: List is empty
    otherValues contains: 2 3 4 6 8
After assign values contains: 2 3 4 6 8
values contains: 2 2 3 3 4 4 6 6 8 8
After remove (4) values contains: 2 2 3 3 6 6 8 8

```

图 20.17 list 标准库类模板用法示例

## 第 18 行

```
std::list< int > values, otherValues;
```

实例化了两个 list 对象,能存储整型值。第 20 行和第 21 行用函数 push\_front 在 values 开头插入整数,函数 push\_front 为 list 和 deque 类特有( vector 类没有此函数)。第 22 行和第 23 行用函数 push\_back 在 values 尾部插入整数。记住,函数 push\_back 适用于所有序列容器。

第 27 行

```
values.sort();
```

使用 list 成员函数 sort 以递增顺序排列 list 中的元素。注意,它不同于标准模板库中的算法 sort。函数 sort 还有另外一个版本允许程序员提供二元判断函数,该函数带两个参数(取 list 中的值),然后对这两个参数进行比较并返回布尔值,显示结果。这个函数用于确定 list 中元素的排列顺序。这个版本特别适用于存储指针(非数值)的 list。注意:图 20.28 演示了一元判断函数。一元判断函数只带一个参数,然后使用参数进行比较并返回表明结果的布尔值。

第 34 行

```
values.splice(values.end(), otherValues);
```

用 list 函数 splice 将 otherValues 中的元素移除到 values 中第一个参数指定的迭代器位置之前。该函数还有另外两个版本。带 3 个参数的 splice 函数允许从第二个参数指定的容器中移除第 3 个参数指定位置的元素。带 4 个参数的 splice 函数将第二个参数指定的容器中移除最后两个参数指定位置范围内的元素并将这些元素移到第一个参数指定的位置。

在 list otherValues 中插入一些元素,并对 values 与 otherValues 进行排序之后,第 45 行

```
values.merge(otherValues);
```

用 list 成员函数 merge 将 otherValues 中的所有元素以排好的顺序插入 values。在此操作之前,两个列表必须按同样的顺序排。函数 merge 的另一个版本允许程序员提供带两个参数(为 list 中的值)的判断函数并返回一个布尔值。这个判断函数指定 merge 所用的排列顺序。

第 51 行用 list 函数 pop\_front 删除 list 中的第一个元素。第 52 行用函数 pop\_back(所有序列容器都有这一函数)删除中的最后一个元素。

第 56 行

```
values.unique();
```

使用 list 函数 unique 删除 list 中的重复值。执行此操作前,list 应该是按序排列(以便重复值紧挨在一起)。函数 unique 另一个版本允许程序员提供带两个参数(列表中的值)并返回布尔值的判断函数。判断函数确定两个元素是否相等。

第 61 行

```
values.swap(otherValues);
```

用函数 swap(所有容器都有该函数)交换 values 与 otherValues 中的内容。

第 67 行

```
values.assign(otherValues.begin(), otherValues.end());
```

用 list 函数 assign,将 values 中的内容替换为 otherValues 中的内容(由两个迭代器参数指定范围)。函数 assign 的另一个版本则用第二个参数所指定值的副本替换原始内容。函数的第一个参数指定副本数。

第 74 行

```
values.remove(4);
```

使用 list 函数 remove 从 list 中删除数值 4 的所有副本。

### 20.2.3 deque 顺序容器

deque 类集中了 vector 类和 list 类的许多优点。deque(念作“deek”)是“double-ended

queue”的简称,译为 deque。deque 类能使用下标按索引访问以读取和修改元素,这与 vector 类类似。与此同时它也能像 list 类那样在头部和尾部进行有效的插入与删除操作(但 list 类还可以在中间进行有效的插入与删除操作)。deque 类支持随机访问迭代器,因此可用所有标准模板库算法。它最常用的用途是维护先进先出的元素队列。

deque 所需的额外存储空间可以在其两端进行分配,所分配的内存块通常保存为指向这些内存块的指针数组。由于 deque 类不连续的内存布局,因为用于中的迭代器应该比指针(用来遍历 vector 或指针数组)更智能化。

**性能提示 20.12** 一旦为 deque 分配了存储块,在几种实现方法中,都不会释放这些存储块直到 deque 被删除为止。这使得 deque 的操作比反复分配、释放、再分配存储块更有效。但这也意味着 deque 最可能无法有效使用内存(相对于 vector 而言)。

**性能提示 20.13** 在 deque 中部进行插入与删除操作已经过优化尽量减少了要复制的元素个数,以维护 deque 中元素的连续性。

deque 提供了与 vector 相同的基本操作,但也增加了成员函数 push\_front 和 pop\_front,分别在 deque 头部进行插入与删除操作。

图 20.18 演示了 deque 类的特性。记住,图 20.14,20.15 与 20.17 的函数也可用于 deque 类。要使用 deque 类必须包含头文件 <deque>。

```

1 //Fig. 20.18: fig20_18.cpp
2 //Testing Standard Library class deque
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <deque>
9 #include <algorithm>
10
11 int main()
12 {
13     std::deque< double > values;
14     std::ostream_iterator< double > output( cout, " ");
15
16     values.push_front( 2.2 );
17     values.push_front( 3.5 );
18     values.push_back( 1.1 );
19
20     cout << "values contains: ";
21
22     for ( int i = 0; i < values.size(); ++i )
23         cout << values[ i ] << " ";
24
25     values.pop_front();
26     cout << "\nAfter pop_front values contains: ";
27     std::copy ( values.begin(), values.end(), output );
28

```



```

29  values[ 1 ] = 5.4;
30  cout << "\nAfter values[ 1 ] = 5.4 values contains: ";
31  std::copy( values.begin(), values.end(), output );
32  cout << endl;
33  return 0;
34  }

```

输出结果:

```

values contains: 3.5 2.2 1.1
After pop_front values contains: 2.2 1.1
After values [ 1 ] = 5.4 values contains: 2.2 5.4

```

图 20.18 deque 标准库类模板用法示例

第 13 行

```
std::deque<double> values;
```

实例化了一个 deque,用于能够存储 double 值。第 16 行到第 18 行分别用函数 push\_front 和 push\_back 在 deque 头部和尾部插入元素。记住,函数 push\_back 适用于所有序列容器,函数 push\_front 则只适用于 list 类与 deque 类。

第 22 行的 for 循环结构

```

for (int i=0; i<values.size(); ++i)
    cout << values[i] << " ";

```

用下标操作符取出 deque 中每个元素的值作为输出。注意条件中用了函数 size 以确保访问没有超出 deque 边界。

第 25 行用函数 pop\_front 演示了如何移除 deque 中的第一个元素。记住,函数 pop\_front 只适用于 list 类与 deque 类(不适用于 vector 类)。

第 29 行

```
values[1] = 5.4;
```

用下标操作符创建了一个左值。这样即可将值赋给 deque 中的任意一个元素。

## 20.3 关联容器

标准模板库中的关联容器提供了直接访问以便能够通过“关键字”(常称为“查找关键字”)来存储和检索元素。关联容器有 4 种:multiset, set, multimap 和 map。每种容器中,关键字都是按顺序排列的。遍历容器即按此顺序进行。multiset 类与 set 类提供了操纵值集合的操作(值即关键字,也就是说无其他值与关键字关联)。multiset 与 set 的主要差别是 multiset 允许关键字重复 set 则不允许。multimap 与 map 类提供了操纵与关键字相联的值(这些值有时也被称为“映射值”)的操作。multimap 与 map 的主要差别 multimap 允许映射值有重复关键字, map 则不允许。除了图 20.2 所有容器的成员函数外,所有的关联容器也支持其他几个成员函数,包括 find, lower\_bound, upper\_bound 和 count。关联容器以及通用于关联容器的成员函数示例参见后文。

### 20.3.1 multiset 关联容器

multiset 关联容器能够快速存储和检索关键字。multiset 允许关键字重复。元素排列的

顺序由“比较器函数对象”确定。例如,在整数 `multiset` 中,元素能够使用比较器函数对象 `less<int>` 排序关键字,以升序元素排列。关联容器中的关键字数据类型必须支持基于指定比较器函数对象的比较,比如使用 `less<int>` 排序的关键字必须支持操作符 `<`。如果容器的关键字为程序员定义的数据类型,这些类型了也必须能支持相应的比较操作。`multiset` 支持双向迭代器(但不支持随机访问迭代器)。

**性能提示 20.14** 考虑到性能,`multiset` 与 `set` 的典型实现方法为所谓的红黑二叉查找树。通过这种内部表示法,二叉查找树很容易达到平衡,这样可以尽量减少平均查找时间。

图 20.19 演示了升序排列的整型值 `multiset`,要使用 `multiset` 类必须包含头文件 `<set>`。容器 `multiset` 与 `set` 提供相同的成员函数。

```

1 //Fig. 20.19; fig20_19.cpp
2 //Testing Standard Library class multiset
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <set>
9 #include <algorithm>
10
11 int main()
12 {
13     const int SIZE = 10;
14     int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
15     typedef std::multiset< int, std::less< int > > ims;
16     ims intMultiset;    //ims for "integer multiset"
17     std::ostream_iterator< int > output( cout, " " );
18
19     cout << "There are currently " << intMultiset.count( 15 )
20         << " values of 15 in the multiset\n";
21     intMultiset.insert( 15 );
22     intMultiset.insert( 15 );
23     cout << "After inserts, there are "
24         << intMultiset.count( 15 )
25         << " values of 15 in the multiset\n";
26
27     ims::const_iterator result;
28
29     result = intMultiset.find( 15 ); //find returns iterator
30
31     if ( result != intMultiset.end() ) //if iterator not at end
32         cout << "Found value 15\n";    //found search value 15
33
34     result = intMultiset.find( 20 );
35
36     if ( result == intMultiset.end() ) //will be true hence
37         cout << "Did not find value 20\n"; //did not find 20
38

```

```

39  intMultiset.insert( a, a + SIZE ); //add array a to multiset
40  cout << "After insert intMultiset contains:\n";
41  std::copy( intMultiset.begin(), intMultiset.end(), output );
42
43  cout << "\nLower bound of 22: "
44        << *( intMultiset.lower_bound( 22 ) );
45  cout << "\nUpper bound of 22: "
46        << *( intMultiset.upper_bound( 22 ) );
47
48  std::pair< ims::const_iterator, ims::const_iterator > p;
49
50  p = intMultiset.equal_range( 22 );
51  cout << "\nUsing equal_range of 22"
52        << "\n  Lower bound: " << *( p.first )
53        << "\n  Upper bound: " << *( p.second );
54  cout << endl;
55  return 0;
56 |

```

输出结果:

```

There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset
Found value 15
Did not find value 20
After insert intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100
Lower bound of 22: 22
Upper bound of 22: 30
Using equal_range of 22
  Lower bound: 22
  Upper bound: 30

```

图 20.19 multiset 标准库类模板用法示例

### 第 15 行和第 16 行

```

typedef std::multiset<int, std::less<int>> ims;
ims intMultiset; //ims for "integer multiset"

```

用 typedef 创建了一个新类型名(别名,该名称是函数对象 less<int> 按升序排列的整型 multiset)。然后用这个新类型实例化一个整型 multiset 对象 intMultiset。

**良好编程习惯 20.1** 运用 typedef,可使类型名(如 multiset)较长的代码更易于阅读。

### 第 19 行和第 20 行的输出语句

```

cout << "There are currently " << intMultiset.count( 15 )
<< " values of 15 in the multiset\n ";

```

用函数 count(所有关联函数都有该函数)计算数值 15 当前 multiset 中出现的次数。

### 第 21 行和第 22 行

```

intMultiset.insert(15);
intMultiset.insert(15);

```

用函数 `insert` 3 个版本中的一个,两次将数值 15 添加到 `multiset`。函数 `insert` 的另一个版本将一个迭代器和一个值作为参数,并从迭代器指定的位置开始查找插入点。函数 `insert` 的第 3 个版本将两个指定了数值范围的迭代器作为参数以便把另一个容器中的元素加入 `multiset`。

第 29 行

```
result = intMultiset.find(15);    //find returns iterators
```

用函数 `find`(所有的关联容器都有该函数)在 `multiset` 中定位数值 15。函数 `find` 返回指向最先出现该数值的迭代器或常量迭代器。如果找不到这个值, `find` 将返回其值与 `end` 返回值相等的迭代器或常量迭代器。

第 39 行

```
intMultiset.insert(a,a+SIZE);    //add array a to multiset
```

用函数 `insert`,将数组 `a` 中的元素插入 `multiset`。在第 41 行,算法 `copy` 将 `multiset` 中的元素复制到标准输出中。注意显示的元素是按升序排列的。

第 43 ~ 46 行

```
cout << " \nLower bound of 22; "
<< *(intMultiset.lower_bound(22));
cout << " \nUpper bound of 22; "
<< *(intMultiset.upper_bound(22));
```

用函数 `lower_bound` 与 `upper_bound`(所有的关联容器都有这两个函数)确定数值 22 最早和最晚出现在 `multiset` 的位置之后那个元素的位置。这两个函数都返回指向相应位置的迭代器或常量迭代器。如果在 `multiset` 中找不到数值 22,则返回其值等于 `end` 所返回迭代器的迭代器。

第 48 行

```
std::pair<ims::const_iterator,ims::const_iterator> p;
```

实例化了 `pair` 类的对象 `p`。`pair` 类的对象用于联系数值对。本例中, `pair` 中的对象为两个基于整型 `multiset` 的常量迭代器。`p` 用于存储 `multiset` 函数 `equal_range` 返回的值。函数 `equal_range` 返回包含下边界 `lower_bound` 与上边界 `upper_bound` 操作结果的数值对。`pair` 类包含两个 `public` 数据成员 `first` 与 `second`。

第 50 行

```
p = intMultiset.equal_range(22);
```

用函数 `equal_range` 确定 `multiset` 中数值 22 的下边界 `lower_bound` 与上边界 `upper_bound`。第 52 行和第 53 行分别用 `p.first` 与 `p.second` 获得上边界与下边界,然后用间接引用迭代器输出 `equal_range` 返回地址中存储的值。

### 20.3.2 set 关联容器

`set` 关联容器用于快速存储和检索单一关键字。除 `set` 的关键字必须惟一外, `set` 与 `multiset` 的实现方法相同。所以试图将重复关键字插入 `set` 时,会忽略重复值。这种忽略正是我们需要的 `set` 的数学行为,所以这不会被视作常见编程错误。`set` 支持双向迭代器(但不支持随机访问迭代器)。图 20.20 演示了双精度值的 `set`,要使用 `set` 类,必须包含头文件 `<set>`。

```

1 //Fig.20.20: fig20_20.cpp
2 //Testing Standard Library class set
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <set>
9 #include <algorithm>
10
11 int main()
12 {
13     typedef std::set< double, std::less< double> > double_set;
14     const int SIZE = 5;
15     double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
16     double_set doubleSet( a, a + SIZE );
17     std::ostream_iterator< double> output( cout, " " );
18
19     cout << "doubleSet contains: ";
20     std::copy( doubleSet.begin(), doubleSet.end(), output );
21
22     std::pair< double_set::const_iterator, bool> p;
23
24     p = doubleSet.insert( 13.8 ); //value not in set
25     cout << '\n' << *( p.first )
26         << ( p.second ? " was" : " was not" ) << " inserted";
27     cout << "\ndoubleSet contains: ";
28     std::copy( doubleSet.begin(), doubleSet.end(), output );
29
30     p = doubleSet.insert( 9.5 ); //value already in set
31     cout << '\n' << *( p.first )
32         << ( p.second ? " was" : " was not" ) << " inserted";
33     cout << "\ndoubleSet contains: ";
34     std::copy( doubleSet.begin(), doubleSet.end(), output );
35
36     cout << endl;
37     return 0;
38 }

```

输出结果:

```

doubleSet contains: 2.1 3.7 4.2 9.5
13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

```

图 20.20 set 标准库类模板用法示例

### 第 13 行

```
typedef std::set< double, std::less<double> > double_set;
```

用 typedef 创建了一个新类型名(别名),该别名用于函数对象 less< double> 按升序排列的

一组 double 值。

第 16 行

```
double_set doubleSet(a, a + SIZE);
```

用新类型 double\_set 实例化了对象 doubleSet。构造函数将数组 a 中 a 到 a + SIZE(即整个数组)之间的所有的元素插入 set。第 20 行用算法 copy 输出 set 中的内容。注意在数组 a 中出现了 2 次的数值 2.1 在 doubleSet 中只出现了一次。这是因为 set 不允许出现重复值。

第 22 行

```
std::pair< double_set::const_iterator, bool > p;
```

定义了包含 double\_set 的迭代器与一个布尔值的数值对 pair。这个对象用于存储调用 set 函数 insert 后返回的结果。

第 24 行

```
p = doubleSet.insert(13.8); //value not in set
```

用函数 insert 将数值 13.8 放入 set。返回的数值对 p 包含了指向数值 13.8 的迭代器 p.first 和一个布尔值(如果插入该值,就返回 true;该值已存在于 set 中,则返回 false)。

### 20.3.3 multimap 关联函数

multimap 关联容器用于快速存储和检索关键字及其关联值(常被称为“关键字/值”数值对)。multiset 与 set 所用的许多方法同样适用于 multimap 与 map, multimap 与 map 中的元素为关键字与值的数值对而不是单个的值。在 multimap 与 map 中进行插入操作时,使用包含关键字与值的数值对对象。关键字的排序由比较器函数对象确定。例如,在用整数作为关键字的 multiset 中,关键字能够通过用比较器函数对象 less<int> 按升序排列。在 multimap 中允许关键字重复,因此重复值能与单一的关键字关联。这常称为一对多关系。例如,在信用卡交易处理系统中,一个信用卡账号可能有很多关联的事务。在大学里,一名学生能选修多门课程,同时一名教授也可带多名学生。在军队中,一种军衔(如美国陆军或海军陆战队的二等兵)可以对应很多人。multimap 支持双向迭代器(但不支持随机访问迭代器)。与 multiset 和 set 一样, multimap 也可以实现为红黑折半查找树(其节点为关键字/值数值对)。图 20.21 演示了 multimap 的用法。要使用 multimap 类,必须包含头文件<map>。

```
1 //Fig.20.21: fig20_21.cpp
2 //Testing Standard Library class multimap
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <map>
9
10 int main()
11 {
12     typedef std::multimap< int, double, std::less< int > > mmid;
13     mmid pairs;
14
15     cout << "There are currently " << pairs.count( 15 )
```

```

16         << " pairs with key 15 in the multimap\n";
17     pairs.insert( mmid::value_type( 15, 2.7 ) );
18     pairs.insert( mmid::value_type( 15, 99.3 ) );
19     cout << "After inserts, there are "
20         << pairs.count( 15 )
21         << " pairs with key 15\n";
22     pairs.insert( mmid::value_type( 30, 111.11 ) );
23     pairs.insert( mmid::value_type( 10, 22.22 ) );
24     pairs.insert( mmid::value_type( 25, 33.333 ) );
25     pairs.insert( mmid::value_type( 20, 9.345 ) );
26     pairs.insert( mmid::value_type( 5, 77.54 ) );
27     cout << "Multimap pairs contains:\nKey\tValue\n";
28
29     for ( mmid::const_iterator iter = pairs.begin();
30         iter != pairs.end(); ++iter )
31         cout << iter ->first << '\t'
32             << iter ->second << '\n';
33
34     cout << endl;
35     return 0;
36 }

```

输出结果:

There are currently 0 pairs with key 15 in the multimap

After inserts, there are 2 pairs with key 15

Multimap pairs contains:

Key

|    |        |
|----|--------|
| 5  | 77.54  |
| 10 | 22.22  |
| 15 | 2.7    |
| 15 | 99.3   |
| 20 | 9.345  |
| 25 | 33.333 |
| 30 | 111.11 |

图 20.21 multimap 标准库类模板用法示例

**性能提示 20.15** multimap 类能够有效地定位与某个指定关键字关联的所有值。

第 12 行

```
typedef std::multimap<int, double, std::less<int> > mmid;
```

用 typedef 定义了 multimap 类关键字类型为 int、关联值类型为 double、元素按升序排列的别名。第 13 行用新类型实例化了一个 multimap 类 pairs。

第 15 行和第 16 行

```
cout << "There are currently" << pairs.count(15)
    << " pairs with key 15 in the multimap\n";
```

用函数 count 确定关键字 15 的关键字/值数值对的个数。

第 17 行

```
pairs.insert(mmid::value_type(15,2.7));
```

用函数 `insert` 在 `multipmap` 类中增加一个新的关键字/值对。表达式 `mmid::value_type(15, 2.7)` 创建了一个数值对对象,其成员 `first` 是 `int` 类型的关键字 15,成员 `second` 是 `double` 类型的数值 2.7。第 12 行将类型 `mmid::value_type` 定义为 `typedef` 的一部分。

第 29 行的 `for` 循环结构输出 `multipmap` 类中的内容,包括关键字与值。第 31 行和第 32 行

```
cout << iter -> first << '\t'
      << iter -> second << '\n';
```

用 `const_iterator` 访问 `multipmap` 类每一个元素中的成员数值对。注意在输出中关键字是按升序排列的。

### 20.3.4 map 关联容器

`map` 关联容器用于快速存储和检索单一关键字。`map` 中不允许出现重复关键字,因此每个关键字只能有一个值与其关联,这称为一对一映射。例如,一家公司可能使用 `map` 将唯一的雇员编号如 100,200 和 300 同他们的电话分机号码 4321,4115 和 5217 一一关联。用 `map` 指定一个关键字可以快速取得与其关联的值,`map` 通常称为关联数组。在 `map` 的下标操作符 `[]` 中使用关键字即可定位与该关键字关联的数值,可以在 `map` 的任意位置进行插入与删除操作。

图 20.22 所示程序演示了 `map` 的用法,它使用与图 20.21 相同的特性(下标操作符除外)。要想使用 `map` 类,必须包含头文件 `<map>`。第 31 行和第 32 行

```
pairs[25] = 9999.99;
pairs[40] = 8765.43;
```

用了 `map` 类的下标操作。当下标为包含在 `map` 中的关键字时,操作符会返回其关联值的引用。当下标为没有包含在 `map` 中的关键字时,操作将关键字插入映射并返回与该关键字关联的值的引用。第 31 行将关键字 25 的值(在第 19 行指定了其初始值 33.333)替换为新值 9999.99。第 32 行在映射中插入一对关键字/值数值对(也称为“创建关联”)。

```
1 //Fig.20.22; fig20_22.cpp
2 //Testing Standard Library class map
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <map>
9
10 int main()
11 |
12     typedef std::map< int, double, std::less< int > > mid;
13     mid pairs;
14
15     pairs.insert( mid::value_type( 15, 2.7 ) );
16     pairs.insert( mid::value_type( 30, 111.11 ) );
17     pairs.insert( mid::value_type( 5, 1010.1 ) );
18     pairs.insert( mid::value_type( 10, 22.22 ) );
```



```

19  pairs.insert( mid::value_type( 25, 33.333 ) );
20  pairs.insert( mid::value_type( 5, 77.54 ) ); //dupe ignored
21  pairs.insert( mid::value_type( 20, 9.345 ) );
22  pairs.insert( mid::value_type( 15, 99.3 ) ); //dupe ignored
23  cout << "pairs contains: \nKey \tValue \n";
24
25  mid::const_iterator iter;
26
27  for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
28      cout << iter ->first << '\t'
29          << iter ->second << '\n';
30
31  pairs[ 25 ] = 9999.99; //change existing value for 25
32  pairs[ 40 ] = 8765.43; //insert new value for 40
33  cout << " \nAfter subscript operations, pairs contains:"
34      << " \nKey \tValue \n";
35
36  for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
37      cout << iter ->first << '\t'
38          << iter ->second << '\n';
39
40  cout << endl;
41  return 0;
42 }

```

输出结果:

pairs contains:

| Key | Value  |
|-----|--------|
| 5   | 1010.1 |
| 10  | 22.22  |
| 15  | 2.7    |
| 20  | 9.345  |
| 25  | 33.333 |
| 30  | 111.11 |

After subscript operations, pairs contains:

| Key | Value   |
|-----|---------|
| 5   | 1010.1  |
| 10  | 22.22   |
| 15  | 2.7     |
| 20  | 9.345   |
| 25  | 9999.99 |
| 30  | 111.11  |
| 40  | 9765.43 |

图 20.22 map 标准库类模板用法示例

## 20.4 容器适配器

标准模板库提供了3类容器适配器: stack, queue 和 priority\_queue。适配器不是第一类

容器,因为它们不提供实际可存储元素的数据结构,它们也不支持迭代器。适配器类的好处是:程序员能够选择合适的底层数据结构。3 种适配器都提供了成员函数 `push` 和 `pop` 用于在适配器数据结构中插入和删除元素。下面几个小节将介绍容器适配器类示例。

### 20.4.1 stack 类适配器

`stack` 类能够在其底层数据结构的一端进行插入与删除操作(常称为“后进先出”数据结构)。 `stack` 能够用任何一种序列容器(`vector`, `list` 和 `deque` 类容器)来实现。本小节分别用标准库中的每一种序列容器作为底层数据结构来表示堆栈。默认情况下, `stack` 用 `deque` 来实现。 `stack` 的操作包括: `push` 将一个元素插入到栈顶(通过调用底层数据结构的函数 `push_back` 来实现), `pop` 删除栈顶的元素(通过调用底层数据结构的函数 `pop_back` 来实现), `top` 返回栈顶元素的引用(通过调用底层数据结构的函数 `back` 来实现), `empty` 用于确定 `stack` 是否为空(通过调用底层数据结构的函数 `empty` 来实现), `size` 用于获得 `stack` 中元素的个数(通过调用底层数据结构的函数 `size` 来实现)。

**性能提示 20.16** `stack` 的各种常见操作都以内联函数的形式,调用其底层数据结构的相应函数来实现,避免了函数调用引起的开销。

**性能提示 20.17** 要想达到最佳的性能,可用 `deque` 类或 `vector` 类作为 `stack` 类的底层数据结构。

图 20.23 演示了适配器 `stack` 类。要使用 `stack` 类,必须包含头文件 `<stack>`。

```

1 //Fig. 20.23: fig20_23.cpp
2 //Testing Standard Library class stack
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <stack>
9 #include <vector>
10 #include <list>
11
12 template< class T >
13 void popElements( T &s );
14
15 int main()
16 {
17     std::stack< int > intDequeStack;    //deque-based stack
18     std::stack< int, std::vector< int > > intVectorStack;
19     std::stack< int, std::list< int > > intListStack;
20
21     for ( int i = 0; i < 10; ++i ) {
22         intDequeStack.push( i );
23         intVectorStack.push( i );
24         intListStack.push( i );
25     }

```

```

26
27     cout << "Popping from intDequeStack: ";
28     popElements( intDequeStack );
29     cout << "\nPopping from intVectorStack: ";
30     popElements( intVectorStack );
31     cout << "\nPopping from intListStack: ";
32     popElements( intListStack );
33
34     cout << endl;
35     return 0;
36 |
37
38 template< class T >
39 void popElements( T &s )
40 {
41     while ( ! s.empty() ) {
42         cout << s.top() << " ";
43         s.pop();
44     }
45 |

```

输出结果:

```

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0

```

图 20.23 stack 标准库类模板用法示例

第 17 ~ 19 行

```

std::stack<int> intDequeStack;    //default is deque-based
std::stack<int,std::vector<int> > intVectorStack;
std::stack<int,std::list<int> > intListStack;

```

实例化了 3 个整型堆栈。第 17 行指定了一个利用默认 deque 作为底层数据结构的整型堆栈,第 18 行指定了一个利用 vector 作为底层数据结构的整型堆栈,第 19 行指定了一个利用 list 作为底层数据结构的整型堆栈。

第 22 行到第 24 行用函数 push(每种适配器类都有该函数)在每个堆栈的顶部插入一个元素。

第 38 行的函数 popElements 用于弹出栈顶的元素。第 42 行用堆栈函数 top 获取栈顶的元素用于输出,它并不会删除栈顶的元素,第 43 行用函数 pop(每种适配器类都有该函数)删除栈顶元素,不返回任何值。

## 20.4.2 queue 类适配器

queue 类适配器能在底层数据结构的尾部进行插入操作,在头部进行删除操作(常称为“先进先出”的数据结构)。deque 能够使用标准模板库数据结构列表和 deque 来实现。默认情况下,queue 是用 deque 来实现的。queue 的常见操作包括:push 将一个元素插入到 queue 尾部(通过调用底层数据结构的函数 push\_back 实现),pop 删除 queue 头部的元素(通过调

用底层数据结构的函数 `pop_back` 实现), `front` 返回 `queue` 中第一个元素的引用(通过调用底层数据结构的函数 `front` 实现), `back` 返回 `queue` 中最后一个元素的引用(通过调用底层数据结构的函数 `back` 实现), `empty` 用于确定 `queue` 是否为空(通过调用底层数据结构的函数 `empty` 实现), `size` 用于获得 `queue` 中的元素个数(通过调用底层数据结构的函数 `size` 实现)。

**性能提示 20.18** `queue` 的每种常见操作都以内联函数的形式,通过调用其底层数据结构的相应函数实现的,避免了函数调用的开销。

**性能提示 20.19** 要想达到最好的性能,可用 `deque` 类作为 `queue` 的底层数据结构。

图 20.24 演示了 `queue` 类适配器。要使用 `queue` 类,必须包含头文件 `<queue>`。

```

1 //Fig.20.24: fig20_24.cpp
2 //Testing Standard Library adapter class template queue
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <queue>
9
10 int main()
11 |
12     std::queue< double > values;
13
14     values.push( 3.2 );
15     values.push( 9.8 );
16     values.push( 5.4 );
17
18     cout << "Popping from values: ";
19
20     while ( ! values.empty() ) {
21         cout << values.front() << " "; //does not remove
22         values.pop();                  //removes element
23     }
24
25     cout << endl;
26     return 0;
27 |

```

输出结果:

Popping from values: 3.2 9.8 5.4

图 20.24 `queue` 标准库类模板用法示例

### 第 12 行

```
std::queue< double > values;
```

实例化了一个 `queue` 存储双精度值。第 14 ~ 16 行用函数 `push` 将元素添加到 `queue`。第 20 行的 `while` 循环结构用函数 `empty`(所有容器都有该函数)确定 `queue` 是否为空。当 `queue` 中还有元素时,第 21 行使用 `queue` 函数 `front` 来读取(不是删除) `queue` 中的第一个元素用于输

出。第 22 行用函数 `pop()` (所有适配器类都有该函数) 删除 `queue` 中的第一个元素。

### 20.4.3 `priority_queue` 类适配器

`priority_queue` 类能够将元素有序地插入到底层数据结构中以及在底层数据结构头部删除元素。`priority_queue` 可以使用标准模板库数据结构 `vector` 和 `deque` 来实现。默认情况下, `priority_queue` 是以 `vector` 作为底层数据结构来实现的。向 `priority_queue` 中加入元素时, 元素是以优先级来排序的, 优先级最高的元素 (即最大值) 就是第一个从 `priority_queue` 删除的元素。这通常通过始终把最大值 (即优先级最高的元素) 放在数据结构 (这种数据结构称为“堆”) 头部的技术来实现的, 该技术称为“堆排序”。默认情况下, 用比较器函数对象 `less<T>` 来执行元素的比较, 程序员也可提供其他比较器函数。

`priority_queue` 的常见操作包括: `push` 根据优先级, 将一个元素插入 `priority_queue` 的相应位置 (通过调用底层数据结构的函数 `push_back` 实现); `pop` 用于删除 `priority_queue` 中优先级最高的元素 (通过调用底层数据结构的函数 `pop_back` 实现); `top` 用于返回 `priority_queue` 中顶层元素的引用 (通过调用底层数据结构的函数 `front` 实现); `empty` 用于确定 `priority_queue` 是否为空 (通过调用底层数据结构的函数 `empty` 实现); `size` 用于获得 `priority_queue` 中元素的数目 (通过调用底层数据结构的函数 `size` 实现)。

**性能提示 20.20** `priority_queue` 的每种常见操作都是以内联函数的形式, 调用其底层数据结构的相应函数实现的, 避免了函数调用的开销。

**性能提示 20.21** 要想获得最好的性能, 可用 `vector` 类作为的底层数据结构。

图 20.25 演示了 `priority_queue` 类适配器。要想使用 `priority_queue`, 必须包含头文件 `<queue>`。

```

1 //Fig.20.25: fig20_25.cpp
2 //Testing Standard Library class priority_queue
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <queue>
9
10 int main()
11 |
12     std::priority_queue< double > priorities;
13
14     priorities.push( 3.2 );
15     priorities.push( 9.8 );
16     priorities.push( 5.4 );
17
18     cout << "Popping from priorities: ";
19
20     while ( ! priorities.empty() ) {
21         cout << priorities.top() << "

```

```

22     priorities.pop();
23     |
24
25     cout << endl;
26     return 0;
27 }

```

输出结果:

Popping from priorities: 9.8 5.4 3.2

图 20.25 `priority_queue` 标准库类模板用法示例

### 第 12 行

```
std::priority_queue< double > priorities;
```

实例化了一个 `priority_queue`, 该模板用于存储双精度值, 且将 `vector` 类用作 `priority_queue` 的底层数据结构。第 14 ~ 16 行用函数 `push` 将元素增加到 `priority_queue` 中。第 20 行的 `while` 循环结构用函数 `empty` (所有容器都有该函数) 来确定 `priority_queue` 是否为空。`priority_queue` 中还有元素, 第 21 行使用 `priority_queue` 函数 `top` 获得优先级最高的元素以便输出。第 22 行用函数 `pop` (所有适配器类都有该函数) 从物理上删除 `priority_queue` 中优先级最高的元素。

## 20.5 算法

在标准模板库出现之前, 不同提供商之间容器的类库与算法实际上是不兼容的。早期的容器库通常使用继承与多态, 产生了调用虚拟函数的开销。而且早期库将算法作为类的行为内嵌于容器类中。标准模板库则是把算法从容器中分离出来, 简化了新算法的添加, 也使标准模板库很有效率, 并且还避免了调用虚拟函数的开销。在标准模板库中, 容器的元素通过迭代器来访问。

**软件工程知识 20.8** 标准模板库算法不依赖于所操作的容器的具体实现。只要容器 (或数组) 的迭代器满足算法的要求, 算法就能用于任何 C 风格的指针数组与容器 (以及用户自定义的数据结构)。

**软件工程知识 20.9** 算法能够在不改变容器类的情况下, 很轻松地添加到标准模板库中。

### 20.5.1 `fill`, `fill_n`, `generate` 和 `generate_n`

图 20.26 演示了标准库函数 `fill`, `fill_n`, `generate` 和 `generate_n` 的用法。函数 `fill` 与 `fill_n` 将指定范围内的容器元素设置为指定的值。函数 `generate` 与 `generate_n` 用“产生器函数”为指定范围内的容器元素创建值。产生器函数不带参数, 且返回能放入容器的值。

```

1 //Fig. 20.26: fig20_26.cpp
2 //Demonstrating fill, fill_n, generate, and generate_n
3 //Standard Library methods.
4 #include <iostream>
5
6 using std::cout;

```

```

7  using std::endl;
8
9  #include <algorithm>
10 #include <vector>
11
12 char nextLetter();
13
14 int main()
15 {
16     std::vector< char > chars( 10 );
17     std::ostream_iterator< char > output( cout, " " );
18
19     std::fill( chars.begin(), chars.end(), '5' );
20     cout << "Vector chars after filling with 5s:\n";
21     std::copy( chars.begin(), chars.end(), output );
22
23     std::fill_n( chars.begin(), 5, 'A' );
24     cout << "\nVector chars after filling five elements"
25         << " with As:\n";
26     std::copy( chars.begin(), chars.end(), output );
27
28     std::generate( chars.begin(), chars.end(), nextLetter );
29     cout << "\nVector chars after generating letters A-J:\n";
30     std::copy( chars.begin(), chars.end(), output );
31
32     std::generate_n( chars.begin(), 5, nextLetter );
33     cout << "\nVector chars after generating K-O for the"
34         << " first five elements:\n";
35     std::copy( chars.begin(), chars.end(), output );
36
37     cout << endl;
38     return 0;
39 }
40
41 char nextLetter()
42 {
43     static char letter = 'A';
44     return letter ++;
45 }

```

输出结果:

Vector chars after filling with 5s:  
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:  
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:  
A B C D E F G H I J

Vector chars after generating K-O for the first five elements:  
K L M N O F G H I J

图 20.26 标准库函数 fill, fill\_n, generate 和 generate\_n 用法示例

## 第 19 行

```
std::fill(chars.begin(),chars.end(),'5');
```

用函数 `fill` 把字符 '5' 放入 `vector chars` 的每个字符中(从 `chars.begin()` 开始直到 `chars.end()` (但不包括该字符),注意作为第一个和第二个参数提供的迭代器必须是正向迭代器(也就是,它们可同时用于容器的正向输入和输出))。

## 第 23 行

```
std::fill_n(chars.begin(),5,'A');
```

用函数 `fill_n` 将字符 'A' 放入 `vector chars` 的前 5 个元素中。作为第一个参数提供的迭代器至少应该是输出迭代器(即它能够正将输出到容器);第二个参数指定了要填充的元素个数;第三个参数指定了要放入的值。

## 第 28 行

```
std::generate(chars.begin(),chars.end(),nextLetter);
```

用函数 `generate` 将产生器函数 `nextLetter` 的返回值放入到 `vector chars` 从 `chars.begin()` 直到 `chars.end()` (但不包括 `chars.end()`) 之间的所有位置的元素中。第一个和第二个参数中的迭代器至少应为正向迭代器。函数 `nextLetter` (参见第 41 行定义)存放在静态局部变量中的字符 'A' 开始工作。第 44 行

```
return letter ++;
```

递增 `letter` 的值,并返回每次调用 `nextLetter` 时的原始值,然后再递增 `letter` 的值。

## 第 32 行

```
std::generate_n(chars.begin(),5,nextLetter);
```

用函数 `generate_n` 将产生器函数 `nextLetter` 的返回值放入 `vector chars` 内 `chars.begin()` 以后的 5 个元素中。第一个参数中的迭代器至少应为输出迭代器。

20.5.2 `equal`, `mismatch` 和 `lexicographical_compare`

图 20.27 演示了如何用标准库函数 `equal`, `mismatch` 和 `lexicographical_compare` 比较序列值的相等性。

```
1 //Fig.20.27: fig20_27.cpp
2 //Demonstrates standard library functions equal,
3 //mismatch, lexicographical_compare.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11
12 int main()
13 {
14     const int SIZE = 10;
15     int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
16     int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
17     std::vector< int > v1( a1, a1 + SIZE ),
```



```

18         v2( a1, a1 + SIZE ),
19         v3( a2, a2 + SIZE );
20     std::ostream_iterator< int > output( cout, " " );
21
22     cout << "Vector v1 contains: ";
23     std::copy( v1.begin(), v1.end(), output );
24     cout << "\nVector v2 contains: ";
25     std::copy( v2.begin(), v2.end(), output );
26     cout << "\nVector v3 contains: ";
27     std::copy( v3.begin(), v3.end(), output );
28
29     bool result =
30         std::equal( v1.begin(), v1.end(), v2.begin() );
31     cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
32         << " equal to vector v2. \n";
33
34     result = std::equal( v1.begin(), v1.end(), v3.begin() );
35     cout << "Vector v1 " << ( result ? "is" : "is not" )
36         << " equal to vector v3. \n";
37
38     std::pair< std::vector< int >::iterator,
39             std::vector< int >::iterator > location;
40     location =
41         std::mismatch( v1.begin(), v1.end(), v3.begin() );
42     cout << "\nThere is a mismatch between v1 and v3 at "
43         << "location " << ( location.first - v1.begin() )
44         << " \nwhere v1 contains " << *location.first
45         << " and v3 contains " << *location.second
46         << " \n\n";
47
48     char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";
49
50     result = std::lexicographical_compare(
51         c1, c1 + SIZE, c2, c2 + SIZE );
52     cout << c1
53         << ( result ? " is less than " :
54             " is greater than or equal to " )
55         << c2 << endl;
56
57     return 0;
58 |

```

**输出结果:**

```

Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

```

```

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

```

```

There is a mismatch between v1 and v3 at location 4

```

where v1 contains 5 and v3 contains 1000

HELLO is greater than or equal to BYE BYE

图 20.27 标准库函数 `equal`, `mismatch` 和 `lexicographical_compare` 用法示例

#### 第 28 行

```
bool result =
    std::equal(v1.begin(), v1.end(), v2.begin());
```

利用函数 `equal` 比较两列数值的相等性。这两列数值的元素个数不一定要相等(如果长度不等, `equal` 会返回 `false`)。函数 `operator ==` 用于比较元素。在本例中, `vector v1` 从 `v1.begin()` 直到 `v1.end()` (但不包括 `v1.end()`) 的元素与向量 `v2` 中从位置 `v2.begin()` 开始的元素进行比较(本例中 `v1` 等于 `v2`)。3 个迭代器参数起码应为输入迭代器(即能够向前从序列中获取输入)。第 34 行用函数 `equal` 比较 `v1` 与 `v3` 是否相等。

另一个版本的 `equal` 函数将一个二元判断函数作为第 4 个参数。二元判断函数接收用于比较的两个元素, 并返回表明元素是否相等的 `bool` 值。这特别适用于存储对象或值的指针而不是值的序列, 因为这样可以定义一个或多个比较。例如, 可以比较对象 `Employee` 的年龄、社会保险号或住址而不是比较整个对象。可以比较指针指向的内容而不是指针本身的内容(即存储在指针中的地址)。

#### 第 38 ~ 41 行

```
std::pair<std::vector<int>::iterator,
std::vector<int>::iterator> location;
location =
    std::mismatch(v1.begin(), v1.end(), v3.begin());
```

首先实例化了针对整型 `vector` 的迭代器对 `location`, 它用于存储调用函数 `mismatch` 的结果。函数 `mismatch` 比较两列数值并返回一对迭代器(表明每一个序列中不匹配元素的位置)。如果所有元素都匹配, 返回的迭代器对就等于两个序列中的最后一个迭代器。3 个迭代器参数起码应为是输入迭代器。本例中, 为了确定 `vector` 中不匹配的实际位置, 在第 43 行使用了表达式 `location.first - v.begin()`。表达式计算结果为两个迭代器之间元素的个数(这与第 5 章的指针算法类似)。本例中, 计算结果对应于元素号, 因为比较从 `vector` 头部开始。

与函数 `equal` 一样, 函数 `mismatch` 还有另外一个版本, 用于将一个二元判定函数作为第 4 个参数。

#### 第 50 行和第 51 行

```
result = std::lexicographical_compare(
    c1, c1 + SIZE, c2, c2 + SIZE);
```

用函数 `lexicographical_compare` 比较两个字符数组中的内容。函数的 4 个迭代器参数起码应为输入迭代器。指向数组的指针为随机访问迭代器。前两个迭代器参数指定第一个序列中的位置范围。后两个迭代器参数指定第二个序列中的位置范围。当在序列中进行迭代时, 如果第一个序列的元素小于第二序列中相应的元素, 函数返回 `true`。如果第一个序列的元素大于或等于第二个序列中相应的元素, 函数返回 `false`。此函数能够用来按词典顺序排

序序列(这些序列通常包含字符串)。

### 20.5.3 remove, remove\_if, remove\_copy 和 remove\_copy\_if

图 20.28 演示了标准库函数 remove, remove\_if, remove\_copy 和 remove\_copy\_if 从序列中删除值的过程。

```

1 //Fig. 20.28: fig20_28.cpp
2 //Demonstrates Standard Library functions remove, remove_if
3 //remove_copy and remove_copy_if
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11
12 bool greater9( int );
13
14 int main()
15 {
16     const int SIZE = 10;
17     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18     std::ostream_iterator< int > output( cout, " " );
19
20     //Remove 10 from v
21     std::vector< int > v( a, a + SIZE );
22     std::vector< int >::iterator newLastElement;
23     cout << "Vector v before removing all 10s:\n";
24     std::copy( v.begin(), v.end(), output );
25     newLastElement = std::remove( v.begin(), v.end(), 10 );
26     cout << "\nVector v after removing all 10s:\n";
27     std::copy( v.begin(), newLastElement, output );
28
29     //Copy from v2 to c, removing 10s
30     std::vector< int > v2( a, a + SIZE );
31     std::vector< int > c( SIZE, 0 );
32     cout << "\n\nVector v2 before removing all 10s "
33         << "and copying:\n";
34     std::copy( v2.begin(), v2.end(), output );
35     std::remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
36     cout << "\nVector c after removing all 10s from v2:\n";
37     std::copy( c.begin(), c.end(), output );
38
39     //Remove elements greater than 9 from v3
40     std::vector< int > v3( a, a + SIZE );
41     cout << "\n\nVector v3 before removing all elements "
42         << "> 9:\n";
43     std::copy( v3.begin(), v3.end(), output );

```

```

44     newLastElement =
45         std::remove_if( v3.begin(), v3.end(), greater9 );
46     cout << " \nVector v3 after removing all elements"
47         << " \ngreater than 9; \n";
48     std::copy( v3.begin(), newLastElement, output );
49
50     //Copy elements from v4 to c2,
51     //removing elements greater than 9
52     std::vector< int > v4( a, a + SIZE );
53     std::vector< int > c2( SIZE, 0 );
54     cout << " \n\nVector v4 before removing all elements"
55         << " \ngreater than 9 and copying: \n";
56     std::copy( v4.begin(), v4.end(), output );
57     std::remove_copy_if( v4.begin(), v4.end(),
58                         c2.begin(), greater9 );
59     cout << " \nVector c2 after removing all elements"
60         << " \ngreater than 9 from v4; \n";
61     std::copy( c2.begin(), c2.end(), output );
62
63     cout << endl;
64     return 0;
65 }
66
67 bool greater9( int x )
68 {
69     return x > 9;
70 }

```

输出结果:

Vector v before removing all 10s:

12 2 10 4 16 6 14 8 12 10

Vector v after removing all 10s:

2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c after removing all 10s from v2:

2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements  
greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after removing all elements  
greater than 9:

2 4 6 8

Vector v4 before removing all elements  
greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

Vector c2 after removing all elements  
greater than 9 from v4:

2 4 6 8 0 0 0 0 0 0

图 20.28 标准库函数 `remove`, `remove_if`, `remove_copy` 和 `remove_copy_if` 用法示例

## 第 25 行

```
newLastElement = std::remove(v.begin(),v.end(),10);
```

用函数 `remove` 删除 `vector v` 中从 `v.begin()` 到 `v.end()` (不包括 `v.end()`) 值为 10 的元素。前两个迭代器参数至少应为正向迭代器以便算法能够修改序列中的元素。这个函数并不修改 `vector` 的元素个数或删除已删除的元素,而是将所有没有删除的元素移至 `vector` 头部。函数返回指向 `vector` 中未删除的元素中最后一个元素之后那个位置的迭代器,而迭代器所指向位置的元素值并不确定(在本例中,每一个“不确定”位置的值都为 0)。

## 第 35 行

```
std::remove_copy(v2.begin(),v2.end(),c.begin(),10);
```

用函数 `remove_copy` 将 `vector v2` 中从 `v2.begin()` 到 `v2.end()` (不包括 `v2.end()` 在内) 不包含数值 10 的元素复制到向量 `c` 中以位置 `c.begin()` 开始的元素。前两个参数中的迭代器必须为输入迭代器。第 3 个参数必须为输出迭代器以便复制元素能插入复制位置。此函数返回指向复制到向量 `c2` 元素中的最后一个位置的迭代器。注意第 31 行使用了取元素个数和这些元素初始值为参数的 `vector` 构造函数。

## 第 44 行和第 45 行

```
newLastElement =  
std::remove_if(v3.begin(),v3.end(),greater9);
```

用函数 `remove_if` 删除 `vector v3` 从 `v3.begin()` 到 `v3.end()` (不包括 `v3.end()` 在内) 中用户定义的二元判定函数返回 `true` 的所有元素。函数 `greater9` 的定义如第 67 行,如果传递给函数的值大于 9,函数返回 `true`,否则返回 `false`。前两个参数中的迭代器必须为正向迭代器,以便算法能够修改序列中的元素。函数并不修改 `vector` 的元素个数,而是所有没有被删除的元素向 `vector` 头部移动。函数返回指向 `vector` 未被删除的元素中最后一个元素之后那个位置的迭代器。而迭代器所指向位置的元素的值并不确定。

## 第 57 行和第 58 行

```
std::remove_copy_if(v4.begin(),v4.end(),  
c2.begin(),greater9);
```

用函数 `remove_copy_if` 复制 `vector v4` 从 `v4.begin()` 到 `v4.end()` (不包括 `v4.end()` 在内) 中用户定义的二元判定函数返回 `true` 的所有元素。然后这些元素放进 `vector c2` 以位置 `c2.begin()` 开始的元素中。前两个参数中的迭代器必须为输入迭代器。第 3 个参数必须为输出迭代器,以便复制元素能够插入复制位置。此函数返回指向被复制到 `vector c2` 元素中的最后一个位置的迭代器。

20.5.4 `replace`, `replace_if`, `replace_copy` 和 `replace_copy_if`

图 20.29 演示了标准库函数 `replace`, `replace_if`, `replace_copy` 和 `replace_copy_if` 用于替换序列值的过程。

```
1 //Fig. 20.29: fig20_29.cpp  
2 //Demonstrates Standard Library functions replace, replace_if  
3 //replace_copy and replace_copy_if  
4 #include <iostream>
```

```
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11
12 bool greater9( int );
13
14 int main()
15 {
16     const int SIZE = 10;
17     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18     std::ostream_iterator< int > output( cout, " " );
19
20     //Replace 10s in v1 with 100
21     std::vector< int > v1( a, a + SIZE );
22     cout << "Vector v1 before replacing all 10s:\n";
23     std::copy( v1.begin(), v1.end(), output );
24     std::replace( v1.begin(), v1.end(), 10, 100 );
25     cout << "\nVector v1 after replacing all 10s with 100s:\n";
26     std::copy( v1.begin(), v1.end(), output );
27
28     //copy from v2 to c1, replacing 10s with 100s
29     std::vector< int > v2( a, a + SIZE );
30     std::vector< int > c1( SIZE );
31     cout << "\n\nVector v2 before replacing all 10s "
32           << "and copying:\n";
33     std::copy( v2.begin(), v2.end(), output );
34     std::replace_copy( v2.begin(), v2.end(),
35                       c1.begin(), 10, 100 );
36     cout << "\nVector c1 after replacing all 10s in v2:\n";
37     std::copy( c1.begin(), c1.end(), output );
38
39     //Replace values greater than 9 in v3 with 100
40     std::vector< int > v3( a, a + SIZE );
41     cout << "\n\nVector v3 before replacing values greater"
42           << " than 9:\n";
43     std::copy( v3.begin(), v3.end(), output );
44     std::replace_if( v3.begin(), v3.end(), greater9, 100 );
45     cout << "\nVector v3 after replacing all values greater"
46           << " than 9 with 100s:\n";
47     std::copy( v3.begin(), v3.end(), output );
48
49     //Copy v4 to c2, replacing elements greater than 9 with 100
50     std::vector< int > v4( a, a + SIZE );
51     std::vector< int > c2( SIZE );
52     cout << "\n\nVector v4 before replacing all values greater"
53           << " than 9 and copying:\n";
```

```

54     std::copy( v4.begin(), v4.end(), output );
55     std::replace_copy_if( v4.begin(), v4.end(), c2.begin(),
56                           greater9, 100 );
57     cout << "\nVector c2 after replacing all values greater"
58           << "\nthan 9 in v4:\n";
59     std::copy( c2.begin(), c2.end(), output );
60
61     cout << endl;
62     return 0;
63 }
64
65 bool greater9( int x )
66 {
67     return x > 9;
68 }

```

输出结果:

```

Vector v1 before replacing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing all 10s with 100s:
100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
10 2 10 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater
than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater
than 9 in v4:
100 2 100 4 100 6 100 8 100 100

```

图 20.29 标准库函数 `replace`, `replace_if`, `replace_copy` 和 `replace_copy_if` 用法示例

#### 第 24 行

```
std::replace(v1.begin(), v1.end(), 10, 100);
```

用函数 `replace` 将 vector `v1` 从 `v1.begin()` 到 `v1.end()` (不包括 `v1.end()`) 中数值为 10 的元素替换为 100。前两个参数中的迭代器必须为正向迭代器以便算法能够修改序列中的元素。

#### 第 34 行和第 35 行

```
std::replace_copy(v2.begin(), v2.end(),
c1.begin(), 10, 100);
```

用函数 `replace_copy` 复制 vector `v2` 从 `v2.begin()` 到 `v2.end()` (不包括 `v2.end()`) 中已被 100 来替换的数值 10 对应的所有元素。然后这些元素复制到 vector `c1` 中以位置 `c1.begin()`

开始的元素。前两个参数中的迭代器必须为输入迭代器。第 3 个参数必须为输出迭代器,以便复制元素能够插入复制位置。此函数返回指向复制到 vector c2 元素中的最后一个元素之后那个位置的迭代器。

第 44 行

```
std::replace_if(v3.begin(),v3.end(),greater9,100);
```

用函数 `replace_if` 将 vector v3 从 `v3.begin()` 到 `v3.end()` (不包括 `v3.end()`) 中用户定义的二元判定函数返回 true 的所有元素替换为 100。函数 `greater9` 在第 65 行定义,如果传递给函数的值大于 9 函数返回 true,反之返回 false。前两个参数中的迭代器必须为正向迭代器,以便算法能够修改序列中的元素。

第 55 行和第 56 行

```
std::replace_copy_if(v4.begin(),v4.end(),c2.begin(),
greater9,100);
```

用函数 `replace_copy_if` 复制 vector v4 中 `v4.begin()` 到 `v4.end()` (但不包含 `v4.end()`) 之间的所有元素。二元判定函数返回 true 的元素替换为 100,然后这些元素放入 vector c2 中以位置 `c2.begin()` 开始的元素中。前两个参数中的迭代器必须为输入迭代器。第 3 个参数必须为输出迭代器,以便复制元素能够插入复制位置。此函数返回指向被复制到 vector c2 元素中的最后一个元素之后位置的迭代器。

## 20.5.5 数学算法

图 20.30 演示了标准模板库中常用的一些数学算法,其中包括 `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` 和 `transform`。

```
1 //Fig.20.30: fig20_30.cpp
2 //Examples of mathematical algorithms in the Standard Library.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <numeric> //accumulate is defined here
10 #include <vector>
11
12 bool greater9( int );
13 void outputSquare( int );
14 int calculateCube( int );
15
16 int main()
17 {
18     const int SIZE = 10;
19     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20     std::vector< int > v( a1, a1 + SIZE );
21     std::ostream_iterator< int > output( cout, " " );
22
```



```

23     cout << "Vector v before random_shuffle: ";
24     std::copy( v.begin(), v.end(), output );
25     std::random_shuffle( v.begin(), v.end() );
26     cout << "\nVector v after random_shuffle: ";
27     std::copy( v.begin(), v.end(), output );
28
29     int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
30     std::vector< int > v2( a2, a2 + SIZE );
31     cout << "\n\nVector v2 contains: ";
32     std::copy( v2.begin(), v2.end(), output );
33     int result = std::count( v2.begin(), v2.end(), 8 );
34     std::cout << "\nNumber of elements matching 8: " << result;
35
36     result = std::count_if( v2.begin(), v2.end(), greater9 );
37     cout << "\nNumber of elements greater than 9: " << result;
38
39     cout << "\n\nMinimum element in Vector v2 is: "
40           << *( std::min_element( v2.begin(), v2.end() ) );
41
42     cout << "\nMaximum element in Vector v2 is: "
43           << *( std::max_element( v2.begin(), v2.end() ) );
44
45     cout << "\n\nThe total of the elements in Vector v is: "
46           << std::accumulate( v.begin(), v.end(), 0 );
47
48     cout << "\n\nThe square of every integer in Vector v is:\n";
49     std::for_each( v.begin(), v.end(), outputSquare );
50
51     std::vector< int > cubes( SIZE );
52     std::transform( v.begin(), v.end(), cubes.begin(),
53                   calculateCube );
54     cout << "\n\nThe cube of every integer in Vector v is:\n";
55     std::copy( cubes.begin(), cubes.end(), output );
56
57     cout << endl;
58     return 0;
59 }
60
61 bool greater9( int value ) { return value > 9; }
62
63 void outputSquare( int value ) { cout << value * value << " "; }
64
65 int calculateCube( int value ) { return value * value * value; }

```

#### 输出结果:

Vector v before random\_shuffle: 1 2 3 4 5 6 7 8 9 10

Vector v after random\_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10

Number of elements matching 8: 3

Number of elements greater than 9: 3

```

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8

```

图 20.30 常用的标准库数学算法示例

## 第 25 行

```
std::random_shuffle( v.begin(), v.end() );
```

用函数 `random_shuffle`, 对 vector `v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的元素进行随机排序。这个函数含有两个随机访问迭代器参数。

## 第 33 行

```
int result = std::count( v2.begin(), v2.end(), 8 );
```

用函数 `count` 统计 vector `v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间数值为 8 的元素个数。这个函数要求其两个迭代器参数至少为输入迭代器。

## 第 36 行

```
result = std::count_if( v2.begin(), v2.end(), greater9 );
```

用函数 `count_if` 统计 vector `v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间, 能使判断函数 `greater9` 返回 `true` 的元素个数。函数 `count_if` 要求它的两个迭代器参数至少为输入迭代器。

## 第 39 行和第 40 行

```
count << " \n\nMinimum element in Vector v2 is: "
<< *( std::min_element( v2.begin(), v2.end() ) );
```

用函数 `min_element` 查找 vector `v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的最小元素。该函数返回这个最小值的输入迭代器, 如果范围为空, 则返回迭代器本身。该函数要求它的两个迭代器参数至少为输入迭代器。这个函数的第二个版本取比较序列元素的函数为它的第 3 个参数。这个比较函数有两个参数并返回一个布尔值。

**良好编程习惯 20.2** 在调用函数 `min_element` 时, 检查指定的范围是否为空, 以及返回值是否是“界外”迭代器, 是个好习惯。

## 第 42 行和第 43 行

```
count << " \n\nMaximum element in Vector v2 is: "
<< *( std::max_element( v2.begin(), v2.end() ) );
```

用函数 `max_element` 查找 vector `v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的最大元素。该函数返回这个最大值的输入迭代器, 如果范围为空, 则返回迭代器本身。该函数要求它的两个迭代器参数至少为输入迭代器。这个函数的第二个版本取比较序列元素的函数作为它的第 3 个参数。这个比较函数有两个参数并返回一个布尔值。

## 第 45 行和第 46 行

```
cout << "\n\nThe total of the elements in Vector v is: "
<< std::accumulate( v.begin(), v.end(), 0 );
```

用函数 `accumulate` (其模板在头文件 `<numeric>` 中) 累计 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间元素值的总和。该函数要求它的两个迭代器参数至少为输入迭代器。之间的第二个版本用一个通用函数作为其第 3 个参数, 由这个通用函数确定哪个元素的值需进行统计。而且该通用函数必须带两个参数并返回一个值。函数的第一个参数是当前的累计值, 第二个参数是序列中要累计的元素值。例如, 要累计所有元素的平方和, 可以使用函数

```
int sumOfSquares( int accumulator, int currentValue )
|
|
return accumulator + currentValue * currentValue;
|
```

接收前一个参数作为它的第一个参数 (`accumulator`), 将要进行平方运算并将其值加入总值的值作为第二个参数 (`currentValue`)。当调用这个函数时, 对 `currentValue` 进行平方, 并把平方后的值加上 `accumulator` 获得的新累计值作为返回值。

第 49 行

```
std::for_each( v.begin(), v.end(), outputSquare );
```

用函数 `for_each` 对 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的元素应用, 一个通用函数。这个通用函数取当前元素作为参数, 但并不修改它。`for_each` 函数要求它的两个迭代器参数至少为输入迭代器。

第 52 行和第 53 行

```
std::transform( v.begin(), v.end(), cubes.begin(),
calculateCube );
```

用函数 `transform` 对 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的元素应用一个通用函数。这个通用函数 (第 4 个参数) 取当前元素为参数, 但不能修改它, 而且应该返回修改后的值。`transform` 函数要求它的前两个迭代器参数至少为输入迭代器, 第 3 个参数至少为输出迭代器。第 3 个参数指定了修改后的值要放置的地方。注意第 3 个参数可以与第一个参数相同。

## 20.5.6 基本查找与排序算法

图 20.31 中的程序演示了标准库中的一些基本查找和排序算法, 这些算法包括 `find`, `find_if`, `sort` 和 `binary_serch`。

```
1 //Fig. 20.31: fig20_31.cpp
2 //Demonstrates search and sort capabilities.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
```

```
9  #include <vector>
10
11  bool greater10( int value );
12
13  int main()
14  {
15      const int SIZE = 10;
16      int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17      std::vector< int > v( a, a + SIZE );
18      std::ostream_iterator< int > output( cout, " " );
19
20      cout << "Vector v contains: ";
21      std::copy( v.begin(), v.end(), output );
22
23      std::vector< int >::iterator location;
24      location = std::find( v.begin(), v.end(), 16 );
25
26      if ( location != v.end() )
27          cout << "\n\nFound 16 at location "
28              << ( location - v.begin() );
29      else
30          cout << "\n\n16 not found";
31
32      location = std::find( v.begin(), v.end(), 100 );
33
34      if ( location != v.end() )
35          cout << "\n\nFound 100 at location "
36              << ( location - v.begin() );
37      else
38          cout << "\n\n100 not found";
39
40      location = std::find_if( v.begin(), v.end(), greater10 );
41
42      if ( location != v.end() )
43          cout << "\n\nThe first value greater than 10 is "
44              << *location << "\n\nfound at location "
45              << ( location - v.begin() );
46      else
47          cout << "\n\nNo values greater than 10 were found";
48
49      std::sort( v.begin(), v.end() );
50      cout << "\n\nVector v after sort: ";
51      std::copy( v.begin(), v.end(), output );
52
53      if ( std::binary_search( v.begin(), v.end(), 13 ) )
54          cout << "\n\n13 was found in v";
55      else
56          cout << "\n\n13 was not found in v";
57
58      if ( std::binary_search( v.begin(), v.end(), 100 ) )
59          cout << "\n\n100 was found in v";
```

```

60     else
61         cout << "100 was not found in v";
62
63     cout << endl;
64     return 0;
65 }
66
67 bool greater10( int value ) { return value > 10; }

```

输出结果:

```

Vector v contains: 10 2 17 5 16 8 13 11 20 7
Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v

```

图 20.31 标准库中的一些基本查找与排序算法示例

#### 第 24 行

```
location = std::find( v.begin(), v.end(), 16 );
```

用函数 `find` 查找 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间数值为 16 的元素,该函数要求它的两个迭代器参数至少为输入迭代器。函数返回一个输入迭代器,这个迭代器要么是第一个数值为 16 的元素,要么指向序列的末端以表明未找到符合条件的元素。

#### 第 40 行

```
location = std::find_if( v.begin(), v.end(), greater10 );
```

用函数 `find_if` 查找 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间第一个使一元判定函数 `greater10` 返回 `true`。函数 `greater10` 定义于第 67 行,它带一个整型参数,并返回一个布尔值,表明这个整型参数是否大于 10。函数 `find_if` 要求它的两个迭代器参数至少为输入迭代器。函数返回一个输入迭代器,这个迭代器要么是第一个使判定函数返回 `true` 的元素,要么指向序列的末端以表明未找到符合条件的元素。

#### 第 49 行

```
std::sort( v.begin(), v.end() );
```

用函数 `sort` 对 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的元素进行升序排序。这个函数要求它的两个迭代器参数为随机迭代器。这个函数的第二个版本使用一个二元判定函数作为它的第 3 个参数,这个判定函数取序列中两个元素的值作为两个参数,并返回一个布尔值,以表明排序顺序(如果返回值为 `true`,则就表明这两个比较元素是以升序排列的)。

**常见编程错误 20.5** 试图用迭代器而不用随机迭代器排序容器是语法错误。函数 `sort` 需要使用随机迭代器。

## 第 53 行

```
if ( std::binary_search( v.begin(), v.end(), 13 ) )
```

用函数 `binary_search` 决定数值 13 是否位于 `vector v` 中 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的元素对应的值之间。在这个操作之前我们必须升序排序序列。函数 `binary_search` 要求它的两个迭代器参数至少为正向迭代器。函数返回一个布尔值以表明数值是否在序列中查到该数值。这个函数的第二个版本取一个二元判定函数作为它的第 4 个参数,判断函数由序列中两个元素的值作为它的两个参数,并返回一个布尔值。如果要比较的两个元素符合指定的排序顺序,判定函数就返回 `true`。

20.5.7 `swap`, `iter_swap` 和 `swap_ranges`

图 20.32 演示了函数 `swap`, `iter_swap` 和 `swap_ranges` 的用法,这些函数用于交换元素。

```
1 //Fig. 20.32: fig20_32.cpp
2 //Demonstrates iter_swap, swap and swap_ranges.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9
10 int main()
11 {
12     const int SIZE = 10;
13     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14     std::ostream_iterator< int > output( cout, " " );
15
16     cout << "Array a contains:\n";
17     std::copy( a, a + SIZE, output );
18
19     std::swap( a[ 0 ], a[ 1 ] );
20     cout << "\nArray a after swapping a[0] and a[1] "
21         << "using swap;\n";
22     std::copy( a, a + SIZE, output );
23
24     std::iter_swap( &a[ 0 ], &a[ 1 ] );
25     cout << "\nArray a after swapping a[0] and a[1] "
26         << "using iter_swap;\n";
27     std::copy( a, a + SIZE, output );
28
29     std::swap_ranges( a, a + 5, a + 5 );
30     cout << "\nArray a after swapping the first five elements\n"
31         << "with the last five elements;\n";
32     std::copy( a, a + SIZE, output );
33
34     cout << endl;
35     return 0;
36 }
```

输出结果:

```
Array a contains:
1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements;
6 7 8 9 10 1 2 3 4 5
```

图 20.32 函数 swap、iter\_swap 和 swap\_ranges 用法示例

第 19 行

```
std::swap( a[ 0 ], a[ 1 ] );
```

用函数 swap 交换两个值。本例中,交换了数组 a 中的第一个元素和第二个元素的值。swap 函数的两个参数是对两个要交换值的引用。

第 24 行

```
std::iter_swap( &a[ 0 ], &a[ 1 ] );
```

用函数 iter\_swap 交换两个元素。iter\_swap 函数带有两个正向迭代器参数(本例中,是数组元素的指针)并且交换迭代器所指向元素的值。

第 29 行

```
std::swap_ranges( a, a + 5, a + 5 );
```

用函数 swap\_ranges 把 a 开始到 a+5(但不包括 a+5)之间的元素与 a+5 开始的元素进行交换。这个函数带有 3 个正向迭代器参数。前两个参数指定第一个序列中要交换元素的范围,第 3 个参数指定第二个序列要交换元素中开始元素的迭代器。本例中,两个要交换的序列值在同一个数组,但也可在不同的数组或容器中。

## 20.5.8 copy\_backward, merge, unique 和 reverse

图 20.33 演示了标准库函数 copy\_backward, merge, unique 和 reverse 的用法。

```
1 //Fig.20.33: fig20_33.cpp
2 //Demonstrates miscellaneous functions: copy_backward, merge,
3 //unique and reverse.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11
12 int main()
13 {
14     const int SIZE = 5;
15     int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
16     int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
```

```

17  std::vector< int > v1( a1, a1 + SIZE );
18  std::vector< int > v2( a2, a2 + SIZE );
19
20  std::ostream_iterator< int > output( cout, " " );
21
22  cout << "Vector v1 contains: ";
23  std::copy( v1.begin(), v1.end(), output );
24  cout << "\nVector v2 contains: ";
25  std::copy( v2.begin(), v2.end(), output );
26
27  std::vector< int > results( v1.size() );
28  std::copy_backward( v1.begin(), v1.end(), results.end() );
29  cout << "\n\nAfter copy_backward, results contains: ";
30  std::copy( results.begin(), results.end(), output );
31
32  std::vector< int > results2( v1.size() + v2.size() );
33  std::merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
34             results2.begin() );
35  cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
36  std::copy( results2.begin(), results2.end(), output );
37
38  std::vector< int >::iterator endLocation;
39  endLocation =
40      std::unique( results2.begin(), results2.end() );
41  cout << "\n\nAfter unique results2 contains:\n";
42  std::copy( results2.begin(), endLocation, output );
43
44  cout << "\n\nVector v1 after reverse: ";
45  std::reverse( v1.begin(), v1.end() );
46  std::copy( v1.begin(), v1.end(), output );
47
48  cout << endl;
49  return 0;
50 }

```

输出结果:

Vector v1 contains: 1 3 5 7 9

Vector v2 contains: 2 4 5 7 9

After copy\_backward results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:  
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:  
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1

图 20.33 标准库函数 `copy_backward`, `merge`, `unique` 和 `reverse` 用法示例

第28行

```
std::copy_backward(v1.begin(), v1.end(), results.end());
```

用函数 `copy_backward` 复制 vector `v1` 中 `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 之间



的元素,并将它们从 `results.end()` 之前的位置逐个向 `vector` 头部方向放入 `vector results`。函数返回复制到 `results` 中元素最后一个之后那个位置的迭代器(即 `vector results` 的开头,因为复制是向后进行的)。元素以与 `v1` 相同的次序放入 `results`。这个函数需要 3 个双向迭代器参数(能够递增或递减,分别向前或向后遍历序列的迭代器)。函数 `copy` 与 `copy_backward` 的主要差别是 `copy` 返回的迭代器指向复制的最后一个元素之后的位置,而 `copy_backward` 返回的迭代器指向复制的最后一个元素所在的位置(实际上是序列中的第一个元素)。此外,`copy` 需要取两个输入迭代器和一个输出迭代器作为参数。

第 33 行和第 34 行

```
std::merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
results2.begin());
```

用函数 `merge` 把两个升序序列合并为另一个升序序列。该函数需要 5 个迭代器参数。前 4 个起码应为输入迭代器,最后一个起码应为输出迭代器。前两个参数指定了第一个有序序列(`v1`)的元素范围,接下来的两个参数指定了第二个有序序列(`v2`)的元素范围,最后一个参数指定了合并后第 3 个序列(`results2`)中的起始位置。函数 `merge` 的另一个版本取指定排列顺序的二元判断函数作为第 5 个参数。

注意,第 32 行用 `v1.size() + v2.size()` 作为元素个数,创建了 `vector results`。用这里所示的函数 `merge` 有一个前提:要求存放合并结果的序列长度要大于或等于要合并的两个序列的长度。如果在合并操作之前不想为合并结果序列分配元素个数,则可用语句

```
vector<int> results2();
merge(v1.begin(),v1.end(),v2.begin(),v2.end(),
back_inserter(results2));
```

参数 `back_inserter(results2)` 使用了用于容器 `results2` 的函数模板 `back_inserter`(包含在头文件 `<iterator>` 中)。`back_inserter` 调用容器的默认函数 `push_back`,在容器末尾插入一个元素。更重要的是,如果将元素插入无其他元素的容器中时,容器的长度会增长。所以并不需要预先知道容器中元素个数。还有其他两个插入函数:`front_inserter` 用于在参数指定的容器头部插入一个元素,`inserter` 用于在第一个参数指定的容器中位于第二个参数指定的迭代器位置之前插入一个元素。

第 39 行和第 40 行

```
endLocation =
std::unique(results2.begin(),results2.end());
```

对 `results2` 中 `results2.begin()` 到 `results2.end()` (但不包含 `results2.end()`) 之间的元素序列使用函数 `unique`。序列中有重复值时,该函数将只保留一个重复值。函数所带的两个参数至少应为正向迭代器。函数返回指向惟一值序列中最后一个元素之后那个位置的迭代器。函数 `unique` 的另一个版本取指定比较两个元素相等性的二元判断函数作为第三个参数。

第 45 行

```
std::reverse(v1.begin(),v1.end());
```

用函数 `reverse` 逆转 `vector v1` 中 `v1.begin()` 到 `v1.end()` (不包括 `v1.end()`) 之间各元素的顺序。函数所带的两个参数至少应为双向迭代器。

### 20.5.9 inplace\_merge, unique\_copy 和 reverse\_copy

图 20.34 中的程序演示了标准库函数 `inplace_merge`, `unique_copy` 和 `reverse_copy` 的用法(注意:该程序无法在 Borland C++ 中编译)。

```

1 //Fig. 20.34: fig20_34.cpp
2 //Demonstrates miscellaneous functions: inplace_merge,
3 //reverse_copy, and unique_copy.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11 #include <iterator>
12
13 int main()
14 {
15     const int SIZE = 10;
16     int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
17     std::vector< int > v1( a1, a1 + SIZE );
18
19     std::ostream_iterator< int > output( cout, " " );
20
21     cout << "Vector v1 contains: ";
22     std::copy( v1.begin(), v1.end(), output );
23
24     std::inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
25     cout << "\nAfter inplace_merge, v1 contains: ";
26     std::copy( v1.begin(), v1.end(), output );
27
28     std::vector< int > results1;
29     std::unique_copy( v1.begin(), v1.end(),
30                     std::back_inserter( results1 ) );
31     cout << "\nAfter unique_copy results1 contains: ";
32     std::copy( results1.begin(), results1.end(), output );
33
34     std::vector< int > results2;
35     cout << "\nAfter reverse_copy, results2 contains: ";
36     std::reverse_copy( v1.begin(), v1.end(),
37                       std::back_inserter( results2 ) );
38     std::copy( results2.begin(), results2.end(), output );
39
40     cout << endl;
41     return 0;
42 }

```

输出结果:

Vector v1 contains: 1 3 5 7 9 1 3 5 7 9

After inplace\_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9  
 After unique\_copy results1 contains: 1 3 5 7 9  
 After reverse\_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1

图 20.34 标准库函数 `inplace_merge`, `unique_copy` 和 `reverse_copy` 用法示例

第 24 行

```
std::inplace_merge(v1.begin(), v1.begin() + 5, v1.end());
```

用函数 `inplace_merge` 将两个有序元素序列合并到同一个容器中。本例中, `v1.begin()` 到 `v1.end()` (但不包括 `v1.end()`) 之间的元素与 `v1.begin() + 5` 到 `v1.end()` (不包括 `v1.end()`) 之间的元素合并在一起。该函数的 3 个迭代器参数至少是双向迭代器。它的另一个版本取一个二元判断函数用于比较两个序列中的元素作为第 4 个参数。

第 29 行和第 30 行

```
std::unique_copy(v1.begin(), v1.end(),  
back_inserter(results1));
```

用函数 `unique_copy` 复制了 `v1.begin()` 到 `v1.end()` (不包括 `v1.end()`) 之间元素序列中所有惟一元素。前两个参数至少应为输入迭代器, 最后一个参数至少应为输出迭代器。本例中, 事先不为 `results1` 分配足够的空间以存储从 `v1` 复制过来的元素。相反地, 使用了函数 `back_inserter` (在头文件 `<iterator>` 中定义) 将元素加入 `vector v1` 尾部。函数 `back_inserter` 使用 `vector` 类的功能在 `vector` 中插入元素。因为它是插入元素而不是替换现有元素, 所以 `vector` 能够增长以容纳附加的元素。函数 `unique_copy` 的另一个版本取一个二元判断函数 (用于比较元素的相等性) 作为第 4 个参数。

第 36 行和第 37 行

```
std::reverse_copy(v1.begin(), v1.end(),  
back_inserter(results2));
```

用函数 `reverse_copy` 反向复制了 `v1.begin()` 到 `v1.end()` (不包括 `v1.end()`) 之间元素序列中的所有元素。然后将复制的元素插入 `vector results2`, 这里使用了 `back_inserter` 对象以确保 `vector` 能够增长以容纳相应数目的复制元素。它的前两个参数至少是双向迭代器, 第 3 个参数至少是输出迭代器。

## 20.5.10 集合操作

图 20.35 演示了用于操作有序值集合的标准库函数 `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` 和 `set_union`。为了演示标准库函数能够用于数组和容器, 本例只使用了数组 (记住, 指向数组的指针为随机访问迭代器)。

```
1 //Fig. 20.35; fig20_35.cpp  
2 //Demonstrates includes, set_difference, set_intersection,  
3 //set_symmetric_difference and set_union.  
4 #include <iostream>  
5  
6 using std::cout;  
7 using std::endl;  
8
```

```
9  #include <algorithm>
10
11  int main()
12  {
13      const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
14      int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15      int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
16      int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
17      std::ostream_iterator< int > output( cout, " " );
18
19      cout << "a1 contains: ";
20      std::copy( a1, a1 + SIZE1, output );
21      cout << "\na2 contains: ";
22      std::copy( a2, a2 + SIZE2, output );
23      cout << "\na3 contains: ";
24      std::copy( a3, a3 + SIZE2, output );
25
26      if ( std::includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
27          cout << "\na1 includes a2";
28      else
29          cout << "\na1 does not include a2";
30
31      if ( std::includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
32          cout << "\na1 includes a3";
33      else
34          cout << "\na1 does not include a3";
35
36      int difference[ SIZE1 ];
37      int *ptr = std::set_difference( a1, a1 + SIZE1,
38                                     a2, a2 + SIZE2, difference );
39      cout << "\nset_difference of a1 and a2 is: ";
40      std::copy( difference, ptr, output );
41
42      int intersection[ SIZE1 ];
43      ptr = std::set_intersection( a1, a1 + SIZE1,
44                                  a2, a2 + SIZE2, intersection );
45      cout << "\nset_intersection of a1 and a2 is: ";
46      std::copy( intersection, ptr, output );
47
48      int symmetric_difference[ SIZE1 ];
49      ptr = std::set_symmetric_difference( a1, a1 + SIZE1,
50                                           a2, a2 + SIZE2, symmetric_difference );
51      cout << "\nset_symmetric_difference of a1 and a2 is: ";
52      std::copy( symmetric_difference, ptr, output );
53
54      int unionSet[ SIZE3 ];
55      ptr = std::set_union( a1, a1 + SIZE1,
56                           a3, a3 + SIZE2, unionSet );
57      cout << "\nset_union of a1 and a3 is: ";
58      std::copy( unionSet, ptr, output );
59      cout << endl;
```

```
60     return 0;
61 }
```

输出结果:

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
a1 includes a2
a1 does not include a3
set_difference of a1 and a2 is: 1 2 3 9 10
set_intersection of a1 and a2 is: 4 5 6 7 8
set_symmetric_difference of a1 and a2 is: 1 2 3 9 10
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

图 20.35 标准库中的集合操作用法示例

## 第 26 行

```
if (std::includes(a1,a1 + SIZE1,a2,a2 + SIZE2))
```

在 if 结构中调用函数 includes 作为条件。函数 includes 比较两个有序值集合,确定第二个集合中的各个元素是否在第一个集合中。如果在,includes 返回 true,否则 includes 返回 false。前两个迭代器参数至少应为输出迭代器,它们表示第一个值集合。本例中,第一个集合包括 a1 到 a1 + SIZE1 (不包括 a1 + SIZE1) 之间的元素。后两个迭代器参数至少应为输出迭代器,它们表示第二个值集合。本例中,第二个集合包括 a2 直到 a2 + SIZE2 (不包括 a1 + SIZE2) 之间的元素。函数 includes 的另一个版本取一下二元判断函数用于比较两个元素的相等性作为第 5 个参数。

## 第 37 行和第 38 行

```
int *ptr = std::set_difference(a1,a1 + SIZE1,
                              a2,a2 + SIZE2,difference);
```

用函数 set\_difference 确定包含在第一个有序值集合中但不包含在第二个有序值集合中的元素(两个集合必须以升序排列)。这些不同元素被复制到第 5 个参数中(这里为数组 difference)。前两个参数至少应为第一个值集合的输入迭代器,接下来的两个参数至少应为第二个值集合的输入迭代器,第 5 个参数至少应为输出迭代器以表明在何处存放这些不同值。函数返回指向复制到第 5 个参数指定的集合中最后一个元素之后那个位置的迭代器。函数 set\_difference 的另一个版本把二元判断函数(用于表明元素原始排列顺序)作为第 6 个参数。这两个序列必须用相同的函数排序。

## 第 43 行和第 44 行

```
ptr = std::set_intersection(a1,a1 + SIZE1,
                             a2,a2 + SIZE2, intersection);
```

用函数 set\_intersection 确定同时包含在第一个有序值集合和第二个有序值集合中的元素(两个集合必须以升序排列)。两个集合共有的元素被复制到第 5 个参数中(这里为数组 intersection)。前两个参数至少为第一个值集合的输入迭代器,接下来的两个参数至少为第二个值集合的输入迭代器,第 5 个参数至少为输出迭代器以表明在何处存放这些共有的值。函数返回指向复制到第 5 个参数指定的集合中最后一个元素之后那个位置的迭代器。函数

`set_difference` 的另一个版本取二元判定函数(用于表明元素原始排列的顺序)作为第 6 个参数。这两个序列必须用相同的函数排序。

第 49 行和第 50 行

```
ptr = std::set_symmetric_difference(a1, a1 + SIZE1,
                                     a2, a2 + SIZE2, symmetric_difference);
```

用函数 `set_symmetric_difference` 确定包含在第一个有序值集合中但不包含在第二个有序值集合中的元素以及包含在第二个有序值集合中但不包含在第一个有序值集合中的元素(两个集合必须以升序排列)。这些不同的元素被复制到第 5 个参数中(这里为数组 `symmetric_difference`)。前两个参数至少为第一个值集合的输入迭代器,接下来的两个参数至少为第二个值集合的输入迭代器,第 5 个参数至少为输出迭代器以表明在何处存放这些不同的值。函数返回指向复制到第 5 个参数指定的集合中最后一个元素之后那个位置的迭代器。函数 `set_symmetric_difference` 的另一个版本取二元判定函数(用于表明元素原始排列的顺序)作为第 6 个参数。这两个序列必须用相同的函数来排序。

第 55 行和第 56 行

```
ptr = std::set_union(a1, a1 + SIZE1,
                    a2, a2 + SIZE2, unionSet);
```

用函数 `set_union` 创建了包含两个有序值集合中所有元素的集合(两个集合必须以升序排列)。这些元素被复制到第 5 个参数中(这里为数组 `unionSet`)。只能从第一个集合中复制同时出现在两个有序值集合中的元素。前两个参数至少为第一个值集合的输入迭代器,接下来的两个参数至少为第二个值集合的输入迭代器,第 5 个参数至少为输出迭代器以表明在何处存放这些不同的值。函数返回指向复制到第 5 个参数指定的集合中最后一个元素之后那个位置的迭代器。函数 `set_symmetric_difference` 的另一个版本取二元判定函数(用于表明元素原始排列的顺序)作为第 6 个参数。这两个序列必须用相同的函数排序。

### 20.5.11 lower\_bound, upper\_bound 和 equal\_range

图 20.36 演示了标准库函数 `lower_bound`, `upper_bound` 和 `equal_range` 的用法。

```
1 //Fig. 20.36: fig20_36.cpp
2 //Demonstrates lower_bound, upper_bound and equal_range for
3 //a sorted sequence of values.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <algorithm>
10 #include <vector>
11
12 int main()
13 |
14     const int SIZE = 10;
15     int a1[] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
```

```

16  std::vector< int > v( a1, a1 + SIZE );
17  std::ostream_iterator< int > output( cout, " " );
18
19  cout << "Vector v contains:\n";
20  std::copy( v.begin(), v.end(), output );
21
22  std::vector< int >::iterator lower;
23  lower = std::lower_bound( v.begin(), v.end(), 6 );
24  cout << "\n\nLower bound of 6 is element "
25        << ( lower - v.begin() ) << " of vector v";
26
27  std::vector< int >::iterator upper;
28  upper = std::upper_bound( v.begin(), v.end(), 6 );
29  cout << "\n\nUpper bound of 6 is element "
30        << ( upper - v.begin() ) << " of vector v";
31
32  std::pair< std::vector< int >::iterator,
33            std::vector< int >::iterator > eq;
34  eq = std::equal_range( v.begin(), v.end(), 6 );
35  cout << "\n\nUsing equal_range;\n"
36        << "    Lower bound of 6 is element "
37        << ( eq.first - v.begin() ) << " of vector v";
38  cout << "\n    Upper bound of 6 is element "
39        << ( eq.second - v.begin() ) << " of vector v";
40
41  cout << "\n\nUse lower_bound to locate the first point\n"
42        << "at which 5 can be inserted in order";
43  lower = std::lower_bound( v.begin(), v.end(), 5 );
44  cout << "\n    Lower bound of 5 is element "
45        << ( lower - v.begin() ) << " of vector v";
46
47  cout << "\n\nUse upper_bound to locate the last point\n"
48        << "at which 7 can be inserted in order";
49  upper = std::upper_bound( v.begin(), v.end(), 7 );
50  cout << "\n    Upper bound of 7 is element "
51        << ( upper - v.begin() ) << " of vector v";
52
53  cout << "\n\nUse equal_range to locate the first and\n"
54        << "last point at which 5 can be inserted in order";
55  eq = std::equal_range( v.begin(), v.end(), 5 );
56  cout << "\n    Lower bound of 5 is element "
57        << ( eq.first - v.begin() ) << " of vector v";
58  cout << "\n    Upper bound of 5 is element "
59        << ( eq.second - v.begin() ) << " of vector v"
60        << endl;
61  return 0;
62 |

```

输出结果:

```

Vector v contains:
2 2 4 4 4 6 6 6 6 8

```

```

Lower bound of 6 is element 5 of vector v
Upper bound of 6 is element 9 of vector v
Using equal_range:
    Lower bound of 6 is element 5 of vector v
    Upper bound of 6 is element 9 of vector v

Use lower_bound to locate the first point
at which 5 can be inserted in order
    Lower bound of 5 is element 5 of vector v

Use upper_bound to locate the last point
at which 7 can be inserted in order
    Upper bound at 7 is element 9 of vector v

Use equal_range to locate the first and
last point at which 5 can be inserted in order
    Lower bound of 5 is element 5 of vector v
    Upper bound of 5 is element 5 of vector v

```

图 20.36 标准库函数 `lower_bound`、`upper_bound` 和 `equal_range` 用法示例

#### 第 23 行

```
lower = std::lower_bound(v.begin(), v.end(), 6);
```

用函数 `lower_bound`，在有序值集合中确定可插入第 3 个参数的第一个位置，插入后的序列仍然保持升序排列。前两个参数中的迭代器至少为正向迭代器。第 3 个参数为确定其下边界 (lower bound) 的值。函数返回指向插入点的正向迭代器。函数 `lower_bound` 的另一个版本取二元判断函数 (用于表明元素原始排列顺序) 作为第 4 个参数。

#### 第 28 行

```
upper = std::upper_bound(v.begin(), v.end(), 6);
```

用函数 `upper_bound` 在有序值集合中确定能插入第 3 个参数的最后一个位置，插入后序列仍然保持升序排列。前两个参数中的迭代器至少为正向迭代器。第 3 个参数为确定其上边界 (upper bound) 的值。函数返回指向插入点的正向迭代器。函数 `upper_bound` 的另一个版本取二元判断函数 (用于表明元素原始排列顺序) 作为第 4 个参数。

#### 第 34 行

```
eq = std::equal_range(v.begin(), v.end(), 6);
```

用函数 `equal_range` 返回包含了执行下边界与上边界操作所得结果的数值对。前两个参数中的迭代器至少为正向迭代器。第 3 个参数为确定其等值范围 (equal range) 的值。函数返回正向迭代器数值对，其第一个成员 (eq.first) 与第二个成员 (eq.second) 分别表示下边界与上边界。

函数 `lower_bound`、`upper_bound` 和 `equal_range` 通常用于定位有序序列中的插入点。第 43 行用 `lower_bound` 定位数值 5 在 `v` 中的第一个插入点。第 49 行用 `upper_bound` 定位数值 7 在 `v` 中的最后一个插入点。第 55 行用 `equal_range` 定位数值 5 在 `v` 中的第一个与最后一个插入点。

## 20.5.12 堆排序

图 20.37 演示了执行堆排序算法的标准库函数。堆排序算法用于将数组中的元素排列



成一种特殊二叉树(称为堆)。堆的重要特性为:其最大的元素总是处于堆顶,并且二叉树中任一节点的子节点值总是小于或等于该节点的值。这种堆通常称为“最大堆”(maxheap)。堆排序算法通常在计算机课程“数据结构”与“算法”中讲到。

```

1 //Fig.20.37: fig20_37.cpp
2 //Demonstrating push_heap, pop_heap, make_heap and sort_heap.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10
11 int main()
12 |
13     const int SIZE = 10;
14     int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
15     int i;
16     std::vector< int > v( a, a + SIZE ), v2;
17     std::ostream_iterator< int > output( cout, " " );
18
19     cout << "Vector v before make_heap: \n";
20     std::copy( v.begin(), v.end(), output );
21     std::make_heap( v.begin(), v.end() );
22     cout << " \nVector v after make_heap: \n";
23     std::copy( v.begin(), v.end(), output );
24     std::sort_heap( v.begin(), v.end() );
25     cout << " \nVector v after sort_heap: \n";
26     std::copy( v.begin(), v.end(), output );
27
28     //perform the heapsort with push_heap and pop_heap
29     cout << " \n \nArray a contains: ";
30     std::copy( a, a + SIZE, output );
31
32     for ( i = 0; i < SIZE; ++i ) |
33         v2.push_back( a[ i ] );
34         std::push_heap( v2.begin(), v2.end() );
35         cout << " \nv2 after push_heap(a[ " << i << " ]): ";
36         std::copy( v2.begin(), v2.end(), output );
37     }
38
39     for ( i = 0; i < v2.size(); ++i ) |
40         cout << " \nv2 after " << v2[ 0 ] << " popped from heap \n";
41         std::pop_heap( v2.begin(), v2.end() - i );
42         std::copy( v2.begin(), v2.end(), output );
43     }
44
45     cout << endl;
46     return 0;

```

47 |

输出结果:

```

Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40
v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 77 52 3 22 31 1 3
v2 after push_heap(a[8]): 100 77 52 3 22 31 1 3 13
v2 after push_heap(a[9]): 100 77 52 3 22 31 1 3 13 22
v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100

```

图 20.37 执行堆排序的标准库函数用法示例

第 21 行

```
std::make_heap(v.begin(),v.end());
```

用函数 `make_heap` 接收 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的序列值,并创建了用于产生有序排列的堆。两个迭代器参数必须是随机访问迭代器,因此该函数只能用于数组、vector 和 deque。函数 `make_heap` 的另一个版本取一个二元判定函数(用于比较数值)为第 3 个参数。

第 24 行

```
std::sort_heap(v.begin(),v.end());
```

用函数 `sort_heap` 对已经形成堆的 `v.begin()` 到 `v.end()` (但不包括 `v.end()`) 之间的序列值进行排序。两个迭代器参数必须是随机访问迭代器。函数 `make_heap` 的另一个版本取一个二元判断函数(用于比较值)作为第 3 个参数。

#### 第 34 行

```
std::push_heap(v2.begin(),v2.end());
```

用函数 `push_heap` 在堆中增加一个新值。我们一次取数组 `a` 的一个元素,并将它加入 vector `v2` 末尾,然后执行 `push_heap` 操作。如果增加的元素为 vector 中的惟一一个元素, vector 就已经成为堆。每次调用 `push_heap`,它都假定 vector 当前的最后一个元素(即调用 `push_heap` 之前所增加的元素)为要加入堆中的元素, vector 中其他元素已经形成了堆。两个迭代器参数必须是随机访问迭代器。函数 `push_heap` 的另一个版本取一个二元判断函数(用于比较值)作为第 3 个参数。

#### 第 41 行

```
std::pop_heap(v2.begin(),v2.end()-i);
```

用函数 `pop_heap` 删除堆顶元素。此函数假定两个随机访问迭代器参数所指定范围内的元素已经形成了堆。反复删除堆顶元素最终会形成一个有序排列。函数 `pop_heap` 将第一个堆元素(本例中为 `v2.begin()`)与最后一个堆元素(本例中为 `v2.end()-i`)交换,然后确保最后一个元素(不包括最后一个)的元素仍然形成了堆。注意在 `pop_heap` 操作之后的输出中, vector 是以升序排列的。函数 `pop_heap` 的另一个版本取一个二元判断函数(用来比较值)作为第 3 个参数。

### 20.5.13 min 和 max

算法 `min` 与 `max` 分别用于确定两个元素中的最小值与最大值。图 20.38 的程序演示了如何用 `min` 与 `max` 处理整型值和字符值(注意: Microsoft Visual C++ 并不支持标准模板库算法 `min` 与 `max`,因为它们与微软基础类库 MFC 中的同名函数冲突。MFC 是微软公司用于创建 Windows 应用程序的可重用类。图 20.38 中的程序是在 Borland C++ 中编译的)。

```
1 //Fig. 20.38: fig20_38.cpp
2 //Demonstrating min and max
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9
10 int main()
11 {
12     cout << "The minimum of 12 and 7 is; "
13         << std::min( 12, 7 );
14     cout << " \nThe maximum of 12 and 7 is; "
15         << std::max( 12, 7 );
16     cout << " \nThe minimum of 'G' and 'Z' is; "
```

```

17         << std::min( 'G', 'Z' );
18     cout << " \nThe maximum of 'G' and 'Z' is: "
19         << std::max( 'G', 'Z' ) << endl;
20     return 0;
21 }

```

输出结果:

```

The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z

```

图 20.38 算法 min 与 max 用法示例

## 20.5.14 本章没有讨论到的算法

图 20.39 列出了本章没有讨论到的算法。

| 算法                  | 说明                                                                                                                                                                   |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| inner_product       | 计算两列数值的内积。即取两列数值中的对应元素并进行相乘,然后把所有的乘积结果加在一起                                                                                                                           |
| adjacent_difference | 从一个序列的第二个元素开始,计算当前元素与前一个元素的差(用操作符-)并保存结果。前两个输入迭代器参数指定了容器中元素的范围,第三个输出迭代器参数指定了用于保存结果的地方。此算法的另一个版本取一个二元判断函数(用于计算当前元素与前一个元素)作为第4个参数                                      |
| partial_sum         | 计算序列中值的连续和(使用操作符+)。前两个输入迭代器参数指定了容器中元素的范围,第3个输出迭代器参数指定了用于保存结果的地方。此算法的另一个版本取一个二元判断函数(用于计算当前元素与前一个元素)作为第4个参数                                                            |
| nth_element         | 用3个随机访问迭代器分隔一列元素。第一个与最后一个参数表明了元素的范围。第二个参数为分隔元素的位置。执行此算法后,分隔元素左边的所有元素都小于分隔元素,右边的元素都大于或等于分隔元素。此算法的另一个版本取一个二元比较函数作为第4个参数                                                |
| partition           | 与nth_element相似,但它需要功能稍弱的双向迭代器,使其比nth_element更灵活。函数partition需要两个双向迭代器指定容器中元素的范围。第3个参数为用于分隔的一元判定函数,以便序列中返回true的所有元素都在返回false的所有元素左边。函数返回一个指向序列中第一个使判断函数为false的元素的双向迭代器 |
| stable_partition    | 除了判断函数返回true以及返回false的元素仍按原来的顺序排列外,此算法与partition相同                                                                                                                   |
| next_permutation    | 序列的后一个词法置换                                                                                                                                                           |
| prev_permutation    | 序列的前一个词法置换                                                                                                                                                           |
| rotate              | 使用3个正迭代器参数移动的位数为第二个与第一个参数的差。将第一个和最后一个参数指定的序列循环移位,例如,序列1,2,3,4,5循环移动2位后的结果为4,5,1,2,3                                                                                  |
| rotate_copy         | 除了将循环移结果保存到第4个参数(一个输出迭代器)指定的其他序列中,此算法与rotate相同。原始序列与保存结果的序列必须具有相同的元素个数                                                                                               |

(续表)

| 算法                             | 说明                                                                                                                    |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>adjacent_find</code>     | 此算法返回一个输入迭代器,用来表明第一个出现在序列中两个相同且相邻的值。如果没有相同且相邻的元素,返回的迭代器将指向序列末尾                                                        |
| <code>partial_sort</code>      | 用3个随机访问迭代器排序序列。第一个与最后一个参数指明了元素序列。第二个参数指明了序列中要排序部分的结束位置。默认情况下,元素用操作符<(也可使用一个二元判定函数)排序。第二个参数指定的元素到序列末尾的排序不确定            |
| <code>partial_sort_copy</code> | 用两个输入迭代器与两个随机访问迭代器排序两个输入迭代器参数指定的部分序列。结果保存在两个随机访问迭代器参数指定的序列中。默认情况下,元素用操作符<(也可使用一个二元判定函数)排序。排序的元素个数为结果元素个数与原始序列元素个数的较小值 |
| <code>stable_sort</code>       | 除了所有等值的顺序不变外,其他与算法 <code>sort</code> 相同                                                                               |

图 20.39 本章尚未讨论到的算法

## 20.6 bitset 类

`bitset`(位集)类使得创建和操纵位集(bit sets)更容易。位集适用于表示标志位的集合。`Bitset` 在编译时长度是固定的。声明

```
bitset<size> b;
```

创建了 `bitset b`,其中所有位的初始值都为0。语句

```
b.set(bitNumber);
```

将 `bitset b` 中的位 `bitNumber` 设置为“开”。表达式 `b.set()` 将 `b` 中所有位都设置为“开”。

语句

```
b.reset(bitNumber);
```

将 `bitset b` 中的位 `bitNumber` 设置为“关”。表达式 `b.set()` 将 `b` 中所有位都设置为“关”。语句

```
b.flip(bitNumber);
```

“颠倒”了 `bitset b` 中的位 `bitNumber`(如:若为开,flip 则将其设置为关)。表达式 `b.flip()` 翻转 `bitset b` 中所有的位。语句

```
b[bitNumber];
```

返回对 `bitset b` 中位 `bitNumber` 的引用。类似地,语句

```
b.at(bitNumber);
```

先对 `bitNumber` 执行范围检查。然后,如果 `bitNumber` 在范围内,at 函数就返回对这个位的引用;否则,at 抛出 `out_of_range` 异常。语句

```
b.test(bitNumber);
```

先对 `bitNumber` 执行范围检查。然后,如果 `bitNumber` 在范围内,位为开时,test 函数返回 true,位为关时,test 返回 false。若该位不在范围内,test 抛出 `out_of_range` 异常。表达式

```
b.size();
```

返回 `bitset b` 中位的个数。表达式

```
b.count();
```

返回 bitset b 中状态为开的位的个数。表达式

```
b.any();
```

只要在 bitset b 中有位设置为开就返回 true。表达式

```
b.none();
```

在 bitset b 中没有位设置为开返回 true。表达式

```
b == b1
```

```
b != b1
```

分别用于比较两个 bitset 的相等性和不等性。

位赋值操作符  $\&=$ ,  $|=$  和  $\wedge=$  可用于合并 bitset。例如

```
b&=b1;
```

在 bitset b 与 b1 之间执行按位逻辑与。结果存放在 b 中。位逻辑或与位逻辑异或操作也可以通过语句

```
b|=b1;
```

```
b^=b2;
```

执行

表达式

```
b >>= n;
```

将 bitset b 中的位右移 n 位。表达式

```
b <<= n;
```

将 bitset b 中的位左移 n 位。表达式

```
b.to_string()
```

```
b.to_ulong()
```

分别将 bitset b 转换为一个字符串和一个无符号整数。

图 20.40 使用了练习题 4.29 中讨论的查找质数的算法(即 Eratosthenes 筛选法)。这里用 bitset 代替数组实现此算法。图 20.40 中的程序显示了 2 到 1 023 之间的所有质数,然后让用户键入数字,确定它是否为质数。

```
1 //Fig. 20.40: fig20_40.cpp
2 //Using a bitset to demonstrate the Sieve of Eratosthenes.
3 #include <iostream>
4
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 #include <bitset>
14 #include <cmath>
15
```

```

16 int main()
17 {
18     const int size = 1024;
19     int i, value, counter;
20     std::bitset< size > sieve;
21
22     sieve.flip();
23
24     //perform Sieve of Eratosthenes
25     int finalBit = sqrt( sieve.size() ) + 1;
26
27     for ( i = 2; i < finalBit; ++i )
28         if ( sieve.test( i ) )
29             for ( int j = 2 * i; j < size; j += i )
30                 sieve.reset( j );
31
32     cout << "The prime numbers in the range 2 to 1023 are;\n";
33
34     for ( i = 2, counter = 0; i < size; ++i )
35         if ( sieve.test( i ) ) {
36             cout << setw( 5 ) << i;
37
38             if ( ++counter % 12 == 0 )
39                 cout << '\n';
40         }
41
42     cout << endl;
43
44     //get a value from the user to determine if it is prime
45     cout << "\nEnter a value from 1 to 1023 ( -1 to end); ";
46     cin >> value;
47
48     while ( value != -1 ) {
49         if ( sieve[ value ] )
50             cout << value << " is a prime number\n";
51         else
52             cout << value << " is not a prime number\n";
53
54         cout << "\nEnter a value from 2 to 1023 ( -1 to end); ";
55         cin >> value;
56     }
57
58     return 0;
59 }

```

输出结果:

The prime numbers in the range 2 to 1023 are;

|    |     |     |     |     |     |     |     |     |     |     |     |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 2  | 3   | 5   | 7   | 11  | 13  | 17  | 19  | 23  | 29  | 31  | 37  |
| 41 | 43  | 47  | 53  | 59  | 61  | 67  | 71  | 73  | 73  | 83  | 89  |
| 97 | 101 | 103 | 107 | 109 | 113 | 127 | 131 | 137 | 139 | 149 | 151 |

```

157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503
509 521 523 541 547 557 563 569 571 577 587 593
599 601 607 613 617 619 631 641 643 647 653 659
661 671 677 683 691 701 709 719 727 733 739 743
751 757 761 769 773 787 797 809 811 821 823 827
829 839 853 857 859 863 877 881 883 887 907 911
919 929 937 941 947 953 967 971 977 983 991 997
1009 1013 1019 1021

Enter a value from 1 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1

```

图 20.40 bitset 类与 Eratosthenes 筛选法

## 第 20 行

```
std::bitset<size> sieve;
```

创建了长度为 size 的 bitset(本例中 size 为 1 024)。程序中忽略了位置为 0 和 1 的位,默认情况下,bitset 中所有位都设置为“关”。代码

```

//perform sieve of Eratosthenes
int finalBit = sqrt(sieve.size()) + 1;

for (i = 2; i < finalBit; ++i)
    if(sieve.test(i))
        for (int j = 2 * i; j < size; j += i)
            sieve.reset(j);

```

用于确定 2 ~ 1 023 之间的所有质数。整数 finalBit 用于确定算法何时完成,基本算法是:如果一个数除了 1 与其本身外没有别的约数,就可以判断它是质数。从整数 2 开始,一旦知道了某个数为质数,就可以忽略该数所有倍数。2 只能被 1 及其本身所除,所以它为质数。然后就可以忽略 4,6,8 等等。3 只能被 1 及其本身所除,因此可以忽略 3 的所有倍数(记住,所有偶数都被忽略)。

## 20.7 函数对象

函数对象与函数适配器使得标准模板库更加灵活。函数对象包含了一个可以用 operator() 函数调用的函数(该函数从语法和语义上都与 operator() 调用的函数相似)。函数对象与函数适配器在头文件 <functional> 中定义。标准模板库的函数对象能够封装数据。为提高性能标准函数对象都是内联函数。标准模板库中的函数对象如图 20.41 所示。



| 标准模板库中的函数对象                          | 类型 |
|--------------------------------------|----|
| <code>divides &lt;T&gt;</code>       | 算术 |
| <code>equal_to &lt;T&gt;</code>      | 关系 |
| <code>greater &lt;T&gt;</code>       | 关系 |
| <code>greater_equal &lt;T&gt;</code> | 关系 |
| <code>less &lt;T&gt;</code>          | 关系 |
| <code>less_equal &lt;T&gt;</code>    | 关系 |
| <code>logical_and &lt;T&gt;</code>   | 逻辑 |
| <code>logical_not &lt;T&gt;</code>   | 逻辑 |
| <code>logical_or &lt;T&gt;</code>    | 逻辑 |
| <code>minus &lt;T&gt;</code>         | 逻辑 |
| <code>modulus &lt;T&gt;</code>       | 算术 |
| <code>negate &lt;T&gt;</code>        | 算术 |
| <code>not_equal_to &lt;T&gt;</code>  | 关系 |
| <code>plus &lt;T&gt;</code>          | 算术 |
| <code>multiplies &lt;T&gt;</code>    | 算术 |

图 20.41 标准库中的函数对象

图 20.42 中的程序演示了使用数字算法 `accumulate` (在图 20.30 中讨论) 来计算向量中元素的平方和。函数 `accumulate` 的第 4 个参数为一个二元函数对象或一个指向带两个参数并返回一个结果的二元函数的函数指针。函数 `accumulate` 演示了两次: 一次是使用指向一个二元函数的函数指针, 另一次是使用函数对象。

```

1 //Fig. 20.42: fig20_42.cpp
2 //Demonstrating function objects.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <vector>
9 #include <algorithm>
10 #include <numeric>
11 #include <functional>
12
13 //binary function adds the square of its second argument and
14 //the running total in its first argument and
15 //returns the sum
16 int sumSquares( int total, int value )
17     | return total + value * value; |
18
19 //binary function class template which defines an overloaded
20 //operator() that adds the square of its second
21 //argument and the running total in its first argument and
22 //returns the sum
23 template< class T >
24 class SumSquaresClass : public std::binary_function< T, T, T >
25 |

```

```

26 public;
27     const T operator()( const T &total, const T &value )
28     { return total + value * value; }
29 };
30
31 int main()
32 {
33     const int SIZE = 10;
34     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
35     std::vector< int > v( a1, a1 + SIZE );
36     std::ostream_iterator< int > output( cout, " " );
37     int result = 0;
38
39     cout << "vector v contains;\n";
40     std::copy( v.begin(), v.end(), output );
41     result =
42         std::accumulate( v.begin(), v.end(), 0, sumSquares );
43     cout << "\n\nSum of squares of elements in vector v using "
44         << "binary\nfunction sumSquares; " << result;
45
46     result = std::accumulate( v.begin(), v.end(), 0,
47                             SumSquaresClass< int >() );
48     cout << "\n\nSum of squares of elements in vector v using "
49         << "binary\nfunction object of type "
50         << "SumSquaresClass< int >; " << result << endl;
51     return 0;
52 }

```

输出结果:

```

vector v contains;
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in vector v using binary
function sumSquares; 385

Sum of squares of elements in vector v using binary
function object of type SumSquaresClass < int >; 385

```

图 20.42 演示二元函数对象

### 第16行和第17行

```

int sumSquares( int total, int value)
{ return total + value * value; }

```

定义了函数 `sumSquares`, 它首先计算第二个参数 `value` 的平方, 然后把结果加入第一个参数 `total`, 最后返回和。函数 `accumulate` 把它所遍历的容器中每一个元素都传递给本例中 `sumSquares` 的第二个参数。第一次调用 `sumSquares` 时, 第一个参数为 `total` 的初始值(本例中为 `accumulate` 的第3个参数, 即0)。以后调用 `sumSquares` 时就将前一次 `sumSquares` 调用返回的累计和作为第一个参数。`accumulate` 结束时将返回序列中所有元素的平方和。

### 第23行和第29行

```

template <class T>

```

```

class SumSquaresClass:public std::binary_function<T,T,T>
{
public:
    const T operator()(const T&total, const T&value)
        { return total + value * value; }
};

```

定义了继承类 `binary_function` (头文件 `<functional>` 中定义) 的类 `SumSquaresClass`, 然后又定义了带两个参数的重载函数 `operator()`。类 `SumSquaresClass` 用于定义函数对象以便重载函数 `operator()` 执行函数 `sumSquares` 所执行的相同操作。模板 `binary_function` 的 3 个类型参数 (`T`) 分别为 `operator()` 的第一个参数、第二个参数的类型以及返回类型。函数 `accumulate` 把它所遍历的容器中每个元素都传递给 `SumSquaresClass` 类对象的函数 `operator()` 的第二个参数。第一次调用 `operator()` 时, 第一个参数为 `total` 的初始值 (本例中为 `accumulate` 的第 3 个参数, 即 0)。以后调用 `operator()` 时就将前一次 `operator()` 调用返回的累计和作为第一个参数。`accumulate` 结束时返回序列中所有元素的平方和。

第 41 行和第 42 行

```

result =
    std::accumulate(v.begin(),v.end(),0,sumSquares);

```

将指向函数 `sumSquares` 的指针作为最后一个参数, 调用函数 `accumulate`。

第 46 行和第 47 行

```

result = std::accumulate(v.begin(),v.end(),0,
    SumSquaresClass<int>());

```

将 `SumSquaresClass` 类对象作为最后一个参数调用函数 `accumulate`。表达式 `SumSquaresClass<int>()` 创建了 `SumSquaresClass` 类的对象, 然后把该对象传递给函数 `accumulate`, 函数 `accumulate` 再向对象发送消息 (调用函数) `operator()`。上述语句可写为

```

SumSquaresClass<int> sumSquaresObj;
result = accumulate(v.begin(),v.end(),0,sumSquaresObj);

```

第一行定义了一个 `SumSquaresClass` 类的对象; 第二行将该对象传递给函数 `accumulate` 并接收消息 `operator()`。

软件工程知识 20.10 与函数指针不同, 函数对象还可以封装数据。

## 20.8 小结

- 使用标准模板库可以节省大量时间与精力, 产生高质量的程序。
- 在特定的应用程序中, 选择标准库容器通常基于性能方面的考虑。
- 标准模板库容器都是模板, 以便能够与特定应用程序相关的数据类型相适应。
- 标准模板库包含了许多作为容器的常用数据结构, 并提供了很多算法, 用于处理这些容器中的数据。
- 标准模板库容器主要分为 3 类: 序列容器、关联容器与容器适配器。序列容器与关联

容器统称为“第一类容器”。

- 其他 4 类容器称为“近似容器”因为它们与“第一类容器”有相似的功能但却不支持“第一类容器”中的所有功能(数组、字符串、位集和变长数组)。
- `vector` 能够在其结尾快速插入与删除操作,并可以直接访问任意元素,`vector` 支持随机访问迭代器。
- `deque` 能够在头部和结尾快速插入与删除操作,`deque` 支持随机访问迭代器,
- `list` 能够在容器任意位置进行快速插入与删除操作,`list` 类支持双向迭代器。
- `set` 能够快速查找关键字。`set` 不允许关键字重复,`set` 支持双向迭代器。
- `multiset` 能够快速查找关键字。`multiset` 中允许关键字重复,`multiset` 支持双向迭代器。
- `map` 能够快速查找关键字及其相应的“关联”值。`map` 不允许关键字重复(即指明了一对一映射),`map` 支持双向迭代器。
- `multimap` 能够快速查找关键字及其相应的“关联”值。`multimap` 允许关键字重复(即指明了一对多映射),`multimap` 支持双向迭代器。
- `stack` 类为后进先出(LIFO)的数据结构。
- `queue` 为先进先出(FIFO)的数据结构。
- `priority_queue` 为先进先出(FIFO)的数据结构,其中优先级最高的元素总是位于 `priority_queue` 的头部。
- 标准模板库经精心设计以便容器能提供相似的功能。许多操作通用于所有容器,其他操作却只适用于相似容器的子集。这有助于提高标准模板库的可扩充性。
- 标准模板库通常避免用继承与虚拟函数,而用模板进行常规化编程,这样可以达到更好的执行性能。
- 要确保存储在容器中元素的类型支持最小的功能集合(模板的限制),包括复制构造函数、赋值操作以及(对关联容器来说)小于操作符。
- 迭代器用于序列中,这些序列可能在容器中,也可能是输入或输出序列。
- 输入迭代器用于从容器中读取元素。只能正向(即从容器头移动到容器结尾)移动,一次经过一个元素。它只支持单次遍历算法。
- 输出迭代器用于将元素写入到容器中。只能只移动,一次经过一个元素。它只支持单次遍历算法。
- 正向迭代器同时具有输入迭代器与输出迭代器的功能。它支持多次遍历算法。
- 双向迭代器除了具有正向迭代器的功能外,还能向后移动(即从容器结尾移动到容器头)。它支持“多次遍历”算法。
- 随机访问迭代器除具有双向迭代器的功能外,还能直接访问容器中的任意元素,即可以任意向前或向后跳转。
- 每种容器所支持的迭代器类别决定了容器是否能使用标准模板库的特定算法。支持随机访问的迭代器可使用标准模板库中所有的算法。
- 在所有的算法中,数组指针都可以取代迭代器。
- 标准模板库包括了近 70 种算法。改变序列算法会修改它所操作的容器中的元素。不改变序列算法则不会修改它所操纵的容器中的元素。

- 函数 `fill` 与 `fill_n` 将指定范围内的容器元素设置为指定值。
- 函数 `generate` 与 `generate_n` 用“产生器函数”为指定范围内的容器元素产生创建值。
- 函数 `equal` 用于比较两列数值的相等性。
- 函数 `mismatch` 比较两列数值,并返回一对迭代器,以表明每一个序列中不匹配元素的位置。如果所有元素都匹配,返回的迭代器对就包含了两个序列使用函数 `end` 所返回的迭代器。
- 函数 `lexicographical_compare` 比较两个序列中的内容,以确定一个序列是否小于另一个序列(与字符串比较类似)。
- 函数 `remove` 与 `remove_copy` 用于删除序列中与指定值匹配的所有元素。函数 `remove_if` 与 `remove_copy_if` 根据序列中传递给函数的一元判断函数的返回结果 `true`,删除对应的元素。
- 函数 `replace` 与 `replace_copy` 用于替换序列中与指定值匹配的所有元素。函数 `replace_if` 与 `replace_copy_if` 根据序列中传递给函数的一元判断函数的返回结果 `true`,将对应元素替换为新值。
- 函数 `random_shuffle` 随机排序序列中的值。
- 函数 `count` 用于计算序列中与指定值匹配的元素个数。函数 `count_if` 用于计算序列中一元判断函数返回 `true` 的元素个数。
- 函数 `min_element` 用于定位序列中最小的元素。函数 `max_element` 用于定位序列中最大的元素。
- 函数 `accumulate` 用于计算序列中值的总和。它的另一个版本接收指向一个通用函数的指针,该函数带两个参数并返回一个结果,该通用函数用于确定如何累加序列中的元素。
- 函数 `for_each` 将一个通用函数应用于序列中的每一个元素。该函数带一个参数(函数不能修改此参数)并返回空字符。
- 函数 `transform` 将一个通用函数应用于序列中的每一个元素。该函数带一个参数(函数不能修改此参数)并返回改变后的结果。
- 函数 `find` 在序列中定位一个元素。如果找到这个元素,函数返回指向这个元素的迭代器;否则返回指向序列结尾的迭代器。函数 `find_if` 用于定位一元判断函数返回 `true` 的第一个元素。
- 函数 `sort` 用于排序序列中的元素(默认按升序排列,也可提供指明排列顺序的二元判断函数)。
- 函数 `binary_search` 用于确定一个元素是否在有序序列中。
- 函数 `swap` 交换两个值。
- 函数 `iter_swap` 交换两个迭代器所引用的值。
- 函数 `swap_ranges` 交换两个序列中的元素。
- 函数 `copy_backward` 复制一个序列中的元素,并将它们从第二个序列结尾处向序列头部放入第二个序列。
- 函数 `merge` 将两个升序排列合并成第 3 个有序排列。注意 `merge` 也可用于没有排序

的序列中,但合并后不会产生有序序列。

- 函数 `back_inserter` 用容器的默认功能将元素插入到容器结尾。当元素插入无其他元素的容器中时,容器的长度会增长。还有其他两种插入器: `front_inserter` 和 `inserter`。`front_inserter` 将一个元素插入容器(在参数中指定)头部。`inserter` 将一个元素插入到第一个参数所指定的容器中的第二个参数指定的迭代器位置之前。
- 函数 `unique` 删除有序序列中的所有重复值。
- 函数 `reverse` 将序列中元素的顺序翻转过来。
- 函数 `inplace_merge` 将两个有序序列的元素合并到同一个容器中。
- 函数 `unique_copy` 复制序列中的惟一元素。函数 `reverse_copy` 按相反的顺序复制序列中的元素。
- 函数 `includes` 比较两个有序的值集合,确定第二个集合中的所有元素是否在第一个集合中。如果在,函数返回 `true`,否则返回 `false`。
- 函数 `set_difference` 用于确定只包含第一个有序的值集合而不在第二个有序的值集合中的元素(两个有序的值集合必须用相同的比较函数按升序排列)。
- 函数 `set_intersection` 用于确定同时包含在第一个有序值集合和第二个有序值集合中的元素(两个有序的值集合必须用相同的比较函数按升序排列)。
- 函数 `set_symmetric_difference` 用于确定在第一个有序的值集合中但不在第二个有序的值集合中的元素以及不在第一个有序的值集合中但在第二个有序的值集合中的元素(两个有序的值集合必须用相同的比较函数按升序排列)。
- 函数 `set_union` 用于创建包含两个有序集合中所有元素的集合(两个有序的值集合必须用相同的比较函数按升序排列)。
- 函数 `lower_bound` 用于在有序值集合中确定能插入第 3 个参数的第一个位置,插入后序列仍然保持升序状态。
- 函数 `upper_bound` 用于在有序值集合中确定能插入第 3 个参数的最后一个位置,插入后序列仍然保持升序状态。
- 函数 `equal_range` 返回一对正向迭代器,分别包含下边界 `lower_bound` 与上边界 `upper_bound` 的操作结果。
- 堆排序算法是将数组中的元素排列成一种特殊的二叉树称为“堆”。堆的重要特性为:其最大的元素总是处于堆顶,并且二叉树中任一节点的子节点值总是小于或等于这个节点的值。这种堆通常称为最大堆。
- 函数 `make_heap` 用于接收序列值,并创建可产生有序序列的堆。
- 函数 `sort_heap` 对已经排列成堆的序列值进行排序。
- 函数 `push_heap` 在堆中增加一个新值,它假定容器中当前的最后一个元素为要增加到堆中的元素,并且其他元素已经形成了堆。函数 `pop_heap` 用于删除堆顶元素,它假定元素已经形成了堆。
- 函数 `min` 确定两个值中的较小值,函数 `max` 确定两个值中的较大值。
- `bitset` 类使得创建和操纵位的集合更容易,它适用于表示 `bool` 标志位的集合。`bitset` 的长度在编译时已经固定。

## 本章术语

|                                              |                                                                 |
|----------------------------------------------|-----------------------------------------------------------------|
| adapter 适配器                                  | ostream_iterator 输出流迭代器                                         |
| assignment 赋值                                | output iterator 输出迭代器                                           |
| associative array 关联数组                       | platform-independent class libraries<br>与平台无关的类库                |
| associative container 关联容器                   | platform-specific class libraries 平台特有的类库                       |
| bidirectional iterator 双向迭代器                 | priority_queue container adapter class<br>priority_queue 容器适配器类 |
| container 容器                                 | queue container adapter class queue 容器适配器类                      |
| container adapter classes 容器适配器类             | random-access iterator 随机访问迭代器                                  |
| creating an association 创建关联                 | range 范围                                                        |
| deque sequence container deque 序列容器          | reverse iterator 反向迭代器                                          |
| first-class containers 第一类容器                 | reverse the contents of a container<br>翻转容器中的内容                 |
| first-in-first-out (FIFO) 先进先出               | reverse_iterator 反向迭代器                                          |
| forward iterator 正向迭代器                       | sequence 序列                                                     |
| function object 函数对象                         | sequence container 序列容器                                         |
| generic programming 泛型编程                     | sequential container 顺序容器                                       |
| input iterator 输入迭代器                         | set associative container 集合关联容器                                |
| last-in-first-out (LIFO) 后进先出                | sorting algorithm 排序算法                                          |
| list sequence container list 序列容器            | stack container adapter class stack 容器适配器类                      |
| map associative container 映射关联容器             | Standard Template Library (STL) 标准模板库                           |
| multimap associative container multimap 关联容器 | string 字符串                                                      |
| multiset associative container multiset 关联容器 | vector sequence container vector 类序列容器                          |
| mutating-sequence algorithm 改变序列算法           |                                                                 |
| namespace std std 名称空间                       |                                                                 |
| non-mutating-sequence algorithm 不改变序列算法      |                                                                 |
| one-to-one mapping 一对一映射                     |                                                                 |

## 常见编程错误

- 20.1 试图间接引用容器之外的迭代器会导致运行时逻辑错误。尤其不能间接引用或自增 end() 返回的迭代器。
- 20.2 试图为常量容器创建非常量迭代器是语法错误。
- 20.3 vector 不能为空, 否则, 函数 front 与 back 会返回不确定的结果。
- 20.4 删除其中包含指向动态分配对象的指针的元素时没有删除这个对象。
- 20.5 试图用迭代器而不用随机迭代器排序容器是语法错误。函数 sort 需要随机迭代器。

## 良好编程习惯

- 20.1 运用 typedef, 使具有长类型名 (如 multiset) 的代码更易于阅读。
- 20.2 调用函数 min\_element 时, 检查指定的范围是否为空, 以及返回值是否是“界外”迭代器, 是一个好习惯。

## 性能提示

- 20.1 对任何特定的应用来说, 都有几种标准模板库容器可供选择。选择最合适的容器以达

到最高的性能(即速度与长度的平衡)。在标准模板库设计中,有效性是要考虑的关键因素。

- 20.2 标准库提供的功能可以使我们对大量的应用进行有效的操作。但对于有特殊性能要求的应用程序,可能需要自行编写自定义的实现方法。
- 20.3 标准模板库通常不使用继承与虚拟函数而用模板进行常规化编程,以获得更好的执行性能。
- 20.4 了解标准模板库组件。针对特定问题选择最合适的容器,可提高性能和减少内存需求。
- 20.5 在 vector 结尾执行插入操作是高效的。vector 仅仅是在需要时才进行增长以容纳新的元素。在 vector 中间执行插入(或删除)操作代价较高:vector 插入(或删除)点之后所有部分都要进行相应的移动,因为 vector 元素存放在连续的内存空间。这一点与 C 或 C++“原始”数组相似。
- 20.6 需要在容器两端频繁执行插入和删除操作的应用程序通常使用 deque 而不是 vector。虽然可以在两者的头部与尾部进行插入和删除操作,但 vector 类在头部执行插入和删除操作比 deque 类更有效率。
- 20.7 需要在容器的中间和/或两端进行插入和删除操作的应用程序通常使用列表,因为 list 可在任何地方进行插入或删除操作。
- 20.8 使用 vector 容器以达到最好的随机访问性能。
- 20.9 vector 对象能够使用重载的下标操作符[]实现快速地按索引访问,因为它们像 C 或 C++“原始”数组一样存放在连续的内存空间。
- 20.10 一次插入多个元素比一次插入一个元素更有效。
- 20.11 需要更多的空间时,自动将 vector 长度增加一倍,可能会浪费空间。例如,加入一个新元素时,一个拥有 1 000 000 个元素的已满 vector 会将其长度增大到足以容纳 2 000 000 个元素。这会留下 999 999 个元素的空闲空间。程序员可用 resize()更好地控制空间的分配。
- 20.12 一旦为 deque 分配了存储块,在几种实现方法中,都不会释放这些存储块直到删除 deque 为止。这使得 deque 的操作比反复分配、释放、再分配存储块更有效。但这也意味着 deque 最可能无法有效使用内存(相当于 vector 类而言)。
- 20.13 在 deque 中部进行插入与删除操作已经过优化尽量减少了要复制的元素个数,以维护 deque 中元素的连续性。
- 20.14 考虑到性能,multiset 与 set 的典型实现方法为所谓的红黑二叉查找树。在这种内部表示法下,二叉查找树很容易达到平衡,这样可以尽量减小平均查找时间。
- 20.15 multimap 类能够有效定位与某个指定关键字关联的所有值。
- 20.16 stack 类的每一种常见的操作是以内联函数的形式,调用其底层数据结构的相应函数来实现,避免了函数调用引起的开销。
- 20.17 要想达到最好的性能,可使用 deque 类或 vector 类来作为 stack 类的底层数据结构。
- 20.18 queue 的每种常见操作都是以内联函数的形式调用其底层数据结构的相应函数来实现的,避免了函数调用的开销。
- 20.19 要想达到最好的性能,可用 deque 类作为队列的底层数据结构。
- 20.20 priority\_queue 的每种常见操作都是以内联函数的形式调用其底层数据结构的相应



函数来实现的,避免了函数调用的开销。

20.21 要想达到最好的性能,可用 `vector` 类作为 `priority_queue` 的底层数据结构。

### 可移植性提示

20.1 标准模板库必然会成为容器编程的优选方法,使用标准模板库编程可增强代码的可移植性。

20.2 因为标准模板库算法通过迭代器间接操纵容器,所以同一种算法可用于不同的容器。

### 软件工程知识

20.1 标准模板库方法允许编写通用程序,使代码不依赖于底层的容器。这种编程方式被称为“泛型编程”。

20.2 不必事事从头做起,要用 C++ 标准中的可重用组件来编程。标准模板库将许多最常用的数据结构作为容器,并提供了很多算法,以便程序能够利用这些算法访问这些容器中的数据。

20.3 从技术上讲,容器中存储的元素并不需要等于和小于操作符,除非需要比较元素。然而使用模板创建代码时,有些编译器需要模板的所有部分都定义,而有些编译器则只需要在程序中定义实际用到的部分模板。

20.4 使用能满足基本性能需求的“功能最弱”的迭代器有利于产生最具重用性的组件。

20.5 标准模板库的实现很简洁。直到现在,类设计者还是通过编写容器的算法成员函数联系算法与容器,但标准模板库采取了另外一种方式。在标准模板库中,算法与容器是分离的,算法通过迭代器间接操纵容器中的元素。这种分离简化了编写适用于各种容器类的算法。

20.6 标准模板库具有可扩展性。可以直接将算法加入到标准模板库中而不需要对其中的容器作任何改变。

20.7 标准模板库算法能够操纵容器以及基于指针的与 C 语言类似的数组。

20.8 标准模板库算法不依赖于所操作的容器的具体实现。只要容器(或数组)的迭代器满足算法的要求,算法就能用于任何 C 风格的指针数组与容器(以及用户自定义的数据结构)。

20.9 算法能够在不改变容器类的情况下,很容易地添加到标准模板库中。

20.10 与函数指针不同,函数对象还可以封装数据。

### 测试和调试提示

20.1 编写基于指针的数据结构与算法时,必须自己调试与测试,以确保数据结构、类与算法正常工作。在这种低层次上操纵指针很容易导致错误。在这类代码中,内存泄露与非法内存访问的错误最为常见。对大多数程序员和程序来讲,使用标准模板库中预先打包好的、模板化了的数据结构会很有效。使用标准模板库中的代码可节省大量的测试和调试时间。但对大型工程来说,模板的编译时间可能较长。

20.2 常量迭代器的取内容操作(\*)返回对容器元素的常量引用,因此不允许非常量成员函数使用。

- 20.3 用 `const_iterator` 进行的操作返回常量引用以防止修改所操作的容器元素。能用 `const_iterator` 时,尽量不用 `iterator`。这是最低权限原则的又一个实例。
- 20.4 只有随机访问迭代器支持小于操作符 `<`, 所以最好用 `!=` 与 `end()` 测试是否到达容器末尾。

#### 自测题:

- 20.1 (判断正误)标准模板库大量使用了继承与虚拟函数。
- 20.2 标准模板库容器的两种类型为序列容器与\_\_\_\_\_容器。
- 20.3 标准模板库避免用 `new` 与 `delete` 而用\_\_\_\_\_完成各种各样的内存分配与释放。
- 20.4 5 种主要的迭代器类型为\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- 20.5 (判断正误)指针是迭代器的一般形式。
- 20.6 (判断正误)标准模板库算法能够操纵与 C 风格的指针数组。
- 20.7 (判断正误)标准模板库算法被封装为每类容器的成员函数。
- 20.8 (判断正误)算法 `remove` 并不减小元素被删除的 `vector` 的长度的大小。
- 20.9 在标准模板库中,内存分配与释放是通过\_\_\_\_\_完成的。
- 20.10 3 种标准模板库容器适配器为\_\_\_\_\_,\_\_\_\_\_和\_\_\_\_\_。
- 20.11 (判断正误)容器成员函数 `end()` 返回容器中最后一个元素的位置。
- 20.12 标准模板库算法用\_\_\_\_\_间接操作容器中的元素。
- 20.13 算法 `sort` 需要\_\_\_\_\_迭代器。

#### 自测题答案

- 20.1 错误。考虑到性能因素,应避免使用继承与虚拟函数。
- 20.2 关联容器。
- 20.3 分配器。
- 20.4 输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器。
- 20.5 错误。迭代器才是指针的一种形式。
- 20.6 正确。
- 20.7 错误。标准模板库算法并不是成员函数。它们通过迭代器间接操容器。
- 20.8 正确。
- 20.9 分配器。
- 20.10 `stack`, `queue` 和 `priority_queue`。
- 20.11 错误。它返回容器最后一个元素之后的那个位置。
- 20.12 迭代器。
- 20.13 随机访问迭代器。

#### 练习题

- 20.14 编写函数模板 `palindrome`。取一个 `const vector` 作为参数,并根据 `vector` 顺读与倒读是否相同(如包含 1,2,3,2,1 的 `vector` 为回文,顺读与倒读一样,而包含 1,2,3,4 的 `vector` 却不这样)而返回 `true` 或 `false`。
- 20.15 修改图 20.29 中的查找质数算法程序:如果用户输入的不是质数,程序就显示它的因

数。记住,一个质数的因数只有1和它本身。每一个非质数都有其惟一的分解因数。例如,数值54的因数为2,3,3和3。这些因数相乘时,结果为54。对数值54来说,质因数输出应该为2和3。

- 20.16 修改练习题20.15:如果用户输入的不是质数,程序就显示它的质因数以及各质因数在这个数的分解因数中出现的次数。例如,数值54的输出应该为

The unique prime factorization of 54 is: 2 \* 3 \* 3 \* 3

## 因特网与万维网上的标准模板库资源

下面列出了一些因特网与万维网上的标准模板库资源。这些站点包括的内容有:教程、参考资料、常见问题、文章、书籍、访谈和软件。

### 教程

<http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html>

它包含例子、基本原理、组件及扩展标准模板库。可以在此找到标准模板库用法示例代码、有用的解释及有帮助的图表。

[http://web.fiech.net/~hondyg/articles/eff\\_stl.htm](http://web.fiech.net/~hondyg/articles/eff_stl.htm)

它提供了有关标准模板库组件、容器、流与迭代器适配器、变换与选择值、筛选与变换值以及对象。

[http://www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

有助于刚开始学习标准模板库的人。这里详细介绍了标准模板库以及对象空间 STL 工具箱示例。

### 参考资料

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

包括了许多与标准模板库相关的站点列表以及推荐的标准模板库书籍列表。

<http://www.cs.rpi.edu/projects/STL/stl/stl.html>

伦斯莱尔理工大学的标准模板库在线参考主页,包括了对标准模板库的详细解释以及指向其他标准模板库信息资源的链接。

<http://www.sgi.com/Technology/STL/>

硅谷标准模板库程序员指南提供了标准模板库信息相关资源。这里可以下载标准模板库以及查找最新的信息、设计文档和指向其他标准模板库资源的链接。

<http://www.dinkumware.com/refcpp.html>

包括了一些有 ANSI/ISO 标准C++ 库和标准模板库的大量相关信息。

<http://www.roguewave.com/products/xplatform/stdlib/>

Rogue Wave 软件公司的标准C++ 库网页,可以在此下载与其标准C++ 库版本相关的白皮书。

## 常见问题

[ftp://butler.hpl.hp.com/stl/stl.faq](http://butler.hpl.hp.com/stl/stl.faq)

该 ftp 站点提供了标准模板库常见问题集由 Marian Corcoran 维护, Marian Corcoran 是 ANSI 委员会成员和 C++ 专家。

## 文章、书籍和访谈

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

列出了超过 15 个的标准模板库相关站点,并推荐了一些标准模板库相关书籍。

<http://www.byte.com/art/9510/sec12/art3.htm>

Byte 杂志的站点,提供了亚历山大·斯蒂芬(Alexander Stepanov)所写的关于标准模板库的文章。亚历山大·斯蒂芬是标准模板库的创始者之一,他提供了如何在泛型编程中使用标准模板库的信息。

<http://www.sgi.com/Technology/STL/drdoobbs-interview.html>

这是对亚历山大·斯蒂芬的访谈,包含了创建标准模板库时发生的一些趣事。亚历山大·斯蒂芬谈到了标准模板库是如何概念化的,也谈到了泛型编程、缩写词 STL 以及其他一些信息。

## ANSI/ISO C++ 标准

<http://www.ansi.org/>

可在该网站购买 C++ 标准文档。

## 软件

<http://www.cs.rpi.edu/~musser/stl.html>

RPI STL 站点包含了一些信息,如标准模板库与其他的 C++ 库之间的差别,如何编译使用标准模板库的程序,主要的标准模板库包含文件列表,使用标准模板库、标准模板库容器类和标准模板库迭代器的示例程序。同时也列出了与标准模板库兼容的编译器清单以及有关标准模板库源代码及其相关材料的 FTP 站点。

<http://www.mathcs.sjsu.edu/faculty/horstman/safestl.html>

这里可以下载文件 SAFESTL.ZIP,该工具用于在使用了标准模板库的程序中查找错误。

<http://www.objectspace.com/jgl/>

对象空间 Object Space 提供了将标准模板库输出到 Java 的信息。这里可以免费下载他们的标准 < Toolkit > 可移植类库,其中的关键特性包括:容器、迭代器、算法、分配器、字符串和异常。

<http://www.cs.rpi.edu/~wiseb/stl-borland.html>

该站点适合 Borland C++ 编译器用户,作者讲到了注意事项和不兼容之处。

<http://msdn.microsoft.com/visualc/>

Microsoft Visual C++ 的主页。这里提供最新的 Visual C++ 新闻、更新、技术资源、示例以

及可下载的东西。

*<http://www.borland.com/bcppbuilder/>*

Borland C++ Builder 主页。这里提供大量C++ 资源,包括几个C++ 新闻组、最新的产品增强消息、常见问题以及其他使用C++ Builder 进行编程的大量资源。

# 第 21 章 标准C++ 语言的增补

## 学习目标

- 理解和使用布尔数据类型
- 能用映射操作符 `static _ cast`, `const _ cast` 和 `reinterpret _ cast`
- 理解名称空间的概念
- 理解和使用 RTTI 以及两个操作符 `typeid` 和 `dynamic _ cast`
- 理解操作符关键字
- 理解显式构造函数
- 在常量 (`const`) 对象中使用可变数据成员
- 理解和使用指向类成员指针的操作符 `*` 和 `-> *`
- 理解虚拟基类在多重继承中的作用

## 21.1 简介

本章我们将讨论一些标准C++ 特性,其中包括布尔数据类型、映射操作符、名称空间、运行时类型标识(RTTI)和操作符关键字。我们也将讨论类成员指针操作符和虚拟基类。

## 21.2 布尔数据类型

C++ 标准提供了布尔(`bool`)数据类型,它的值可以为 `false` 或 `true`,用于替换原来用 0 代表假和非 0 代表真的表示法。图 21.1 中的程序演示了布尔数据类型的用法。

```
1 //Fig. 21.1: fig21_01.cpp
2 //Demonstrating data type bool.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::cin;
8 using std::boolalpha;
9
10 int main()
11 |
12     bool boolean = false;
13     int x = 0;
14
15     cout << "boolean is " << boolean
16         << " \nEnter an integer: ";
```

```
17  cin >> x;
18
19  cout << "integer " << x << " is"
20      << ( x ? " nonzero " : " zero " )
21      << "and interpreted as ";
22
23  if ( x )
24      cout << "true\n";
25  else
26      cout << "false\n";
27
28  boolean = true;
29  cout << "boolean is " << boolean;
30  cout << "\nboolean output with boolalpha manipulator is "
31      << boolalpha << boolean << endl;
32
33  return 0;
34 }
```

输出结果:

```
boolean is 0
Enter an integer; 22
integer 22 is nonzero and interpreted as true
boolean is 1
boolean output with boolalpha manipulator is true
```

图 21.1 演示布尔基本数据类型

下面将简要说明上述程序。

第 12 行

```
bool boolean = false;
```

声明了布尔型变量 `boolean`, 并将其初始化为 `false`。声明了变量 `x` 并将其初始化为 0。第

15 行

```
cout << "boolean is" << boolean
```

输出变量 `boolean` 的值。这里输出的是数字 0 而不是关键字 `false`。布尔型变量值的默认显示是数字。

第 23 行用 `x` 的值(在第 17 行输入)作为的 `if/else` 语句的条件。如果 `x` 为 0, 条件就为 `false`。反之, 条件为 `true`。注意: 由于负值也是非 0 值, 因此它也成了 `true` 值。

第 28 行把 `true` 赋给了变量 `boolean`。变量 `boolean` 的值(1)在第 29 行输出。默认情况布尔型变量值的输出是数字 0 或 1。这个重载的流插入操作符 `<<` 把布尔型变量值显示为整型数值。

第 30 行和第 31 行

```
cout << "\nboolean output with boolalpha manipulator is "
    << boolalpha << boolean << endl;
```

用流操纵元 `boolalpha` 设置输出流的格式, 设置后的格式把布尔值显示为字符串“`true`”或“`false`”。当然流操纵元 `boolalpha` 也可用于输入流。

指针、整型变量和双精度浮点型变量都能隐式转换为布尔型变量。其中零值转换成 false,而非零值则转换成 true。例如表达式

```
bool dc = false + x*2 - b && true;
```

如果 x 等于 3 且 b 为 true,true 就会赋给布尔型变量 dc。应该注意到赋值表达式的右边计算出来的结果为 5,但是这个值隐式转换为 true。

**良好编程习惯 21.1** 创建表明真假值的状态变量时,用布尔值胜于用整数值。

**良好编程习惯 21.2** 为使程序更清晰,尽量用 true 和 false 代替零和非零。

## 21.3 static\_cast 操作符

C++ 标准草案包含了 4 个可用的强制类型转换操作符,比 C 和 C++ 中使用的旧式类型转换更好。新式的强制类型转换比旧式的强制类型转换功能弱但更具有特性,因此程序员可以更好地控制它用法。类型转换危险,常常是错误的根源,因此新式的类型转换更易于用自动化工具测试和跟踪。新式类型转换另一个好处就是各类型转换函数的用途各不相同,旧式类型转换则是一个类型转换函数可适用于多种用途。

C++ 提供了 static\_cast 操作符完成类型的转换。类型检查在编译时进行。操作符 static\_cast 完成标准转换(例如,null 指针(void \*)转换为字符指针(char \*),整型转换双精度浮点型等等)及它们之间的逆向转换。图 21.2 中的程序演示了操作符 static\_cast 的用法。

```
1 //Fig. 21.2: fig21_02.cpp
2 //Demonstrating the static_cast operator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class BaseClass {
9 public:
10     void f( void ) const { cout << "BASE\n"; }
11 };
12
13 class DerivedClass: public BaseClass {
14 public:
15     void f( void ) const { cout << "DERIVED\n"; }
16 };
17
18 void test( BaseClass * );
19
20 int main()
21 {
22     //use static_cast for a conversion
23     double d = 8.22;
24     int x = static_cast< int >( d);
25
26     cout << "d is " << d << "\nx is " << x << endl;
```



```

27
28   BaseClass * basePtr = new DerivedClass;
29   test( basePtr );    //call test
30   delete basePtr;
31
32   return 0;
33 }
34
35 void test( BaseClass * basePtr )
36 {
37     DerivedClass * derivedPtr;
38
39     //cast base class pointer into derived class pointer
40     derivedPtr = static_cast< DerivedClass * >( basePtr );
41     derivedPtr -> f();    //invoke DerivedClass function f
42 }

```

输出结果:

```

d is 8.22
x is 8
DERIVED

```

图 21.2 操作符 `static_cast` 用法示例

**常见编程错误 21.1** 用操作符 `static_cast` 完成非法的类型转换会产生语法错误。这种非法类型转换包括:把常量类型转换为非常量;在没有公共继承关系的类型之间类型转换指针和引用;类型转换一个类型,同时这种类型没有可完成此类型转换的相应构造函数或转换操作符来完成这种转换。

图 21.2 中的程序定义了两个类: `BaseClass` 和 `DerivedClass`。每个类定义了一个成员函数 `f`。第 23 行和第 24 行

```

double d = 8.22;
int x = static_cast< int >( d );

```

定义并初始化了变量 `d` 和 `x`。操作符 `static_cast` 把变量 `d` 从双精度型(`double`)转换为整型(`int`)。操作符 `static_cast` 可用于执行大多数基本数据类型之间的转换,如整型、浮点型、双精度浮点型等等。

**软件工程知识 21.1** 由于 C++ 标准增加了新的强制类型转换操作符,因此 C 语言式的强制类型转换已经变得过时。

**良好编程习惯 21.3** 尽量用更安全、可靠的 `static_cast` 操作符,避免用 C 语言式的强制类型转换操作符。

第 28 行把一个 `DerivedClass` 对象赋给了 `BaseClass` 类指针 `basePtr`,并在第 29 行把这个指针传递给函数 `test`。指针 `basePtr` 接收传入函数 `test` 的地址。第 37 行声明 `DerivedClass` 类的指针 `derivedPtr`。第 40 行

```

DerivedPtr = static_cast< DerivedClass * >( basePtr );

```

用 `static_cast` 执行 `BaseClass` 类指针到 `DerivedClass` 类指针的向下类型转换。尽管(见第 9 章)从基类指针向派生类指针的向下类型转换较为危险,但 `static_cast` 允许这种转换。函数 `f`

用derivedClass进行调用(第 41 行)。

## 21.4 const \_ cast 操作符

C++ 提供了 const \_ cast 操作符,用于转换 const 和 volatile。图 21.3 中的程序演示了 const \_ cast 的用法。

```

1 //Fig. 21.3; fig21_03.cpp
2 //Demonstrating the const _ cast operator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class ConstCastTest {
9 public:
10     void setNumber( int );
11     int getNumber() const;
12     void printNumber() const;
13 private:
14     int number;
15 };
16
17 void ConstCastTest::setNumber( int num ) { number = num; }
18
19 int ConstCastTest::getNumber() const { return number; }
20
21 void ConstCastTest::printNumber() const
22 {
23     cout << " \nNumber after modification: ";
24
25     //the expression number— would generate compile error
26     //undo const -ness to allow modification
27     const _ cast < ConstCastTest * >( this ) ->number--;
28
29     cout << number << endl;
30 }
31
32 int main()
33 {
34     ConstCastTest x;
35     x.setNumber( 8 ); //set private data number to 8
36
37     cout << "Initial value of number: " << x.getNumber();
38
39     x.printNumber();
40     return 0;
41 }

```

输出结果:

Initial value of number: 8  
 Number after modification: 7

图 21.3 const\_cast 操作符用法示例

第 8~15 行声明了 ConstCastTest 类,该类包含 3 个成员函数和一个 private 成员变量 number,其中有两个成员变量声明为 const。函数 setNumber 设置变量 number 的值;函数 getNumber 返回变量 number 的值。

常量成员函数 printNumber 在第 27 行

```
const_cast < ConstCastTest * >( this )->number;
```

修改了变量 number 的值,如下所示常量成员函数 printNumber 中,this 指针的数据类型为 const ConstCastTest \*。上述语句的前部分用 const\_cast 操作符强制转换 this 指针的“const 特性”。因此在语句的后半部分,this 指针的类型变成了 ConstCastTest \*,这样一来,就可以允许修改变量 number。操作符 const\_cast 不能用于直接强制转换常量变量的“常量特性”。

## 21.5 reinterpret\_cast 操作符

C++ 将操作符 reinterpret\_cast 用于非标准类型转换(例如,从一个指针类型类型转换到另一个指针类型等等)。操作符 reinterpret\_cast 不能用于标准类型之间的转换(例如,从双精度浮点型转换为整型等)。图 21.4 中的程序演示了 reinterpret\_cast 操作符的用法。

```
1 //Fig.21.4: fig21_04.cpp
2 //Demonstrating the reinterpret_cast operator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int x = 120, *ptr = &x;
11
12     cout << *reinterpret_cast<char * >( ptr ) << endl;
13
14     return 0;
15 }
```

输出结果:

x

图 21.4 操作符 reinterpret\_cast 用法示例

上述程序声明了一个整数和一个指针。指针 ptr 初始化为整数变量 x 的地址。第 12 行

```
cout << *reinterpret_cast<char * >( ptr ) << endl;
```

用操作符 reinterpret\_cast 把 ptr(整型指针)转换为字符指针,然后直接引用返回的地址就解除引用。

**测试和调试提示 21.1** 使用 reinterpret\_cast 时,比较容易执行一些危险的操作,这些操作

有可能引发严重的运行时错误。

**可移植性提示 21.1** 用 `reinterpret_cast` 可能使应用程序表现出不同的效果。

## 21.6 名称空间

每一个程序都包含定义在不同范围内的许多标识符。有时一个范围内的变量可能与另一个范围内的变量重名(也就是所谓的冲突),从而产生问题。这样的重名可能在几个层次发生。标识符重名往往出现在第三方厂商提供的库中,这些库使用了同样的名字作为全局标识符(如函数)。发生这种情况时,通常会产生编译错误。

**良好编程习惯 21.4** 尽量避免用下划线(`_`)作为标识符的开头,以免出现链接错误。

C++ 标准试图用名称空间(namespace)来解决这个问题。每个名称空间定义了一个范围,用于放置标识符和变量。要想使用名称空间中的成员,必须用名称空间的名字和二元作用域分辨符(`::`)来限定该成员的名称,格式如下

```
namespace _name::member,
```

或在使用这个成员名称之前使用 `using` 语句;`using` 语句通常放在使用了名称空间成员的文件开头。例如,位于一个源程序文件开始处的语句

```
using namespace namespace_name;
```

便指定了 `namespace_name` 名称空间可用于文件中,无需在每一个成员名前加上 `namespace_name` 和二元作用域分辨符(`::`)。

**良好编程习惯 21.5** 为避免作用域冲突,应尽量在成员前面加上其所在名称空间的名字和二元作用域分辨符(`::`)。

应该知道,并非每个名称空间都是惟一的。有时候由于疏忽,两个第三方提供商可能会为他们各自的名称空间使用相同的名字。图 21.5 中的程序演示了名称空间的用法。

```
1 //Fig. 21.5: fig21_05.cpp
2 //Demonstrating namespaces.
3 #include <iostream>
4 using namespace std; //use std namespace
5
6 int myInt = 98;      //global variable
7
8 namespace Example {
9     const double PI = 3.14159;
10    const double E = 2.71828;
11    int myInt = 8;
12    void printValues();
13
14    namespace Inner { //nested namespace
15        enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
16    }
17 }
18
```

```

19 namespace {           //unnamed namespace
20     double d = 88.22;
21 }
22
23 int main()
24 {
25     //output value d of unnamed namespace
26     cout << "d = " << d;
27
28     //output global variable
29     cout << "\n(global) myInt = " << myInt;
30
31     //output values of Example namespace
32     cout << "\nPI = " << Example::PI << "\nE = "
33         << Example::E << "\nmyInt = "
34         << Example::myInt << "\nFISCAL3 = "
35         << Example::Inner::FISCAL3 << endl;
36
37     Example::printValues(); //invoke printValues function
38
39     return 0;
40 }
41
42 void Example::printValues()
43 {
44     cout << "\nIn printValues:\n" << "myInt = "
45         << myInt << "\nPI = " << PI << "\nE = "
46         << E << "\nd = " << d << "\n(global) myInt = "
47         << myInt << "\nFISCAL3 = "
48         << Inner::FISCAL3 << endl;
49 }

```

输出结果:

```

d = 88.2
(global) myInt = 98
PI = 3.14159
E = 2.71828
myInt = 8
FISCAL3 = 1992

In printValues;
myInt = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) myInt = 98
FISCAL3 = 1992

```

图 21.5 名称空间用法示例

#### 第4行

```
using namespace std;
```

告诉编译器将使用名称空间 `std`。头文件 `<iostream>` 中的内容都在名称空间 `std` 中定义。(注意:很多C++编程人员认为写一条第四行那样的 `using` 语句是一种不良的编程习惯,因为这条语句意味着包括名称空间内的所有内容。)

上述 `using namespace` 语句表明一个名称空间里面的成员间将多次应用于一个程序中。同时该语句允许程序员使用名称空间的所有成员,编写出更简洁的语句。比如,程序员可以写语句

```
cout << "d=" << d;
```

无需写为

```
std::cout << "d=" << d;
```

如果没有第4行,图21.5所示程序中所有 `cout` 和 `endl` 都必须用 `std::` 限定。当然,不管是预定义的名称空间(如 `std`),还是程序员自定义的名称空间,都可以使用 `using namespace` 语句。

第8~17行

```
namespace Example {
    const double PI = 3.14159;
    const double E = 2.71828;
    int myInt = 8;
    void printValue();

    namespace Inner {          //nested namespace
        enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
    }
}
```

用关键字 `namespace` 定义了名称空间 `Example`,名称空间的程序体用两个花括号(`{}`)与外界划定界限。与类的程序体不同,名称空间的程序体不需要用分号结尾。`Example` 名称空间由两个常量(`PI` 和 `E`)、一个整型变量(`myInt`)、一个函数(`printValues`)和一个内嵌的名称空间(`Inner`)组成。注意,名称空间成员 `myInt` 与全局变量 `myInt` 同名。同名变量必须使用不同的作用域,否则就会导致语法错误。一个名称空间可以包含多种类型的成员,如常量、数据、类、内嵌名称空间和函数等等。名称空间的定义必须存在于全局范围内或内嵌于其他名称空间内。

第19~21行

```
namespace {
    double d = 88.22;
}
```

创建了一个未命名的名称空间,这个名称空间包含了一个成员变量 `d`。未命名的名称空间的成员存在于全局名称空间中,程序员可以直接使用它,不必用一个名称空间的名字来限定。与未命名的名称空间成员一样,全局变量也是全局名称空间的一部分,可以在文件中声明它之后的任何地方使用它。

**软件工程知识 21.2** 每个编译单元都有自己惟一的未命名名称空间,也就是说未命名名称空间取代了 `static` 链接说明符。

第 26 行输出 `d` 的值。作为未命名名称空间的一部分,成员 `d` 可以直接使用。第 29 行输出了全局变量 `myInt` 的值。第 32 ~ 35 行

```
cout << " \nPI = " << Example::PI << " p\nE = "
    << Example::E << " \nmyInt = "
    << Example::myInt << " \nFISCAL3 = "
    << Example::Inner::FISCAL3 << endl;
```

输出了 `PI`、`E`、`myInt` 和 `FISCAL3` 的值。`PI`、`E` 和 `myInt` 都是 `Example` 的成员,因而用 `Example::` 加以限定。成员 `myInt` 必须用 `Example::` 加以限定,这是因为有一个全局变量与它同名,若不然会输出与之同名的全局变量的值。`FISCAL3` 是内嵌名称空间 `Inner` 的成员,因而用 `Example::Inner::` 限定。

函数 `printValues` 是名称空间 `Example` 的成员,它可以直接访问同一名称空间(这里是 `Example`)中的其他成员,而不必用名称空间的名字限定这些成员。第 44 行的 `cout` 语句输出 `myInt`、`PI`、`E`、`d` 和全局变量 `myInt` 与 `FISCAL3`。注意,`PI` 和 `E` 没有用 `Example` 限定,变量 `d` 依然可以访问,全局变量 `myInt` 使用了全局范围作用分辨符(`::`)限定,`FISCAL3` 则用 `Inner::` 限定。访问内嵌名称空间中的成员的时,必须用这个内嵌名称空间的名字限定(除非在内嵌名称空间内使用它)。

可以通过关键字 `using` 使用各名称空间中的成员。比如语句

```
using Example::PI;
```

允许在不用名称空间的名字限定的情况下访问 `PI`。这通常适用于需要在程序中频率使用某个名称空间成员时。可以为名称空间指定别名。例如语句

```
namespace CPPHTP3E = CplusplusHowToProgram3E;
```

为 `CplusplusHowToProgram3E` 创建了一个别名 `CPPHTP3E`。

**常见编程错误 21.2** 把 `main` 关键字置于名称空间内是语法错误。

**软件工程知识 21.3** 在大型程序中,最好把每一个实体声明在一个类、函数、块或者名称空间内,以使每个实体的作用更突出。

## 21.7 运行时类型信息 (RTTI)

运行时类型信息提供了一种方法,通过这种方法可以在运行时确定一个对象的类型。本节将讨论两个重要的 RTTI 操作符: `typeid` 和 `dynamic_cast`。图 21.6 中的程序和图 21.7 中的程序分别演示了 `typeid` 和 `dynamic_cast` 的用法。

```
1 //Fig. 21.6: fig21_06.cpp
2 //Demonstrating RTTI capability typeid.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <typeinfo>
9
```

```

10 template < typename T >
11 T maximum( T value1, T value2, T value3 )
12 {
13     T max = value1;
14
15     if ( value2 > max )
16         max = value2;
17
18     if ( value3 > max )
19         max = value3;
20
21     //get the name of the type (i.e., int or double)
22     const char *dataType = typeid( T ).name();
23
24     cout << dataType << "s were compared.\nLargest "
25         << dataType << " is ";
26
27     return max;
28 }
29
30 int main()
31 {
32     int a = 8, b = 88, c = 22;
33     double d = 95.96, e = 78.59, f = 83.89;
34
35     cout << maximum( a, b, c ) << "\n";
36     cout << maximum( d, e, f ) << endl;
37
38     return 0;
39 }

```

输出结果:

```

ints were compared.
Largest int is 88
doubles were compared.
Largest double is 95.96

```

图 21.6 typeid 用法示例

**测试和调试提示 21.2** 为使用 RTTI, 有些编译器需要启用 RTTI 功能。查看编译器的说明文档, 了解 RTTI 的用法。

第 8 行包括了头文件 `<typeinfo>`。要使用 `typeid` 的结果, 必须包括 `<typeinfo>` 头文件。图 21.6 中的程序定义了一个函数模板 `maximum`, 该函数模板接受 3 个数据类型为 `T` 的参数, 并对这 3 个参数进行比较, 并返回最大值。程序中用关键字 `typename` 代替了关键字 `class`。在这里, `typename` 与 `class` 的作用相同。

第 22 行

```
const char * dataType = typeid( T ).name();
```

用函数 `name` 返回一个已经实现了的 C 语言式字符串, 该字符串表示 `T` 的类型。关键字 `typeid` 返回对一个 `type_info` 对象的引用。 `type_info` 对象是一个由系统维护的对象, 用以表



示类型。注意, name 函数返回的字符串是由系统拥有的, 程序员不能删除它。

**良好编程习惯 21.6** 在开关(switch)类型的测试中, 用 typeid 是对 RTTI 的误用, 此时应用虚拟函数。

操作符 dynamic\_cast 保证在运行时执行正确的转换(也就是说编译器不能验证转换是否正确)。操作符 dynamic\_cast 常用于执行向下类型转换, 把基类指针转换为派生类指针映射。图 21.7 中的程序演示了 dynamic\_cast 的用法。

```
1 //Fig. 21.7: fig21_07.cpp
2 //Demonstrating dynamic_cast.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 const double PI = 3.14159;
9
10 class Shape {
11     public:
12         virtual double area() const { return 0.0; }
13 };
14
15 class Circle: public Shape {
16     public:
17         Circle( int r = 1 ) { radius = r; }
18
19         virtual double area() const
20         {
21             return PI * radius * radius;
22         };
23     protected:
24         int radius;
25 };
26
27 class Cylinder: public Circle {
28     public:
29         Cylinder( int h = 1 ) { height = h; }
30
31         virtual double area() const
32         {
33             return 2 * PI * radius * height +
34                 2 * Circle::area();
35         }
36     private:
37         int height;
38 };
39
40 void outputShapeArea( const Shape * );    //prototype
41
```

```

42 int main()
43 {
44     Circle circle;
45     Cylinder cylinder;
46     Shape *ptr = 0;
47
48     outputShapeArea( &circle );    //output circle's area
49     outputShapeArea( &cylinder );  //output cylinder's area
50     outputShapeArea( ptr );        //attempt to output area
51     return 0;
52 }
53
54 void outputShapeArea( const Shape *shapePtr )
55 {
56     const Circle *circlePtr;
57     const Cylinder *cylinderPtr;
58
59     //cast Shape * to a Cylinder *
60     cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
61
62     if ( cylinderPtr != 0 ) //if true, invoke area()
63         cout << "Cylinder's area: " << shapePtr->area();
64     else { //shapePtr does not refer to a cylinder
65
66         //cast shapePtr to a Circle *
67         circlePtr = dynamic_cast< const Circle * >( shapePtr );
68
69         if ( circlePtr != 0 ) //if true, invoke area()
70             cout << "Circle's area: " << circlePtr->area();
71         else
72             cout << "Neither a Circle nor a Cylinder.";
73     }
74
75     cout << endl;
76 }

```

输出结果:

```

Circle's area: 3.14159
Cylinder's area: 12.5664
Neither a Circle nor a Cylinder.

```

图 21.7 dynamic\_cast 用法示例

上述程序定义了一个基类 Shape(第 10 行),它包含一个虚拟函数 area。派生类 Circle(第 5 行)以 public 方式继承了 Shape,派生类 Cylinder(第 27 行)以 public 方式继承了类 Circle。Circle 类和 Cylinder 类都重载了函数 area。

在第 44~46 行的函数 main 中,实例化了一个 Circle 类对象 circle 和一个 Cylinder 类对象 cylinder,并声明了一个 Shape 指针 ptr,同时将其初始化为零。第 48 行到第 50 行调用了 3 次 outputShapeArea 函数(在第 54 行定义)。每次调用函数 outputShapeArea 都将显示下面 3 个结果之一:Circle 的面积、Cylinder 的面积或表示该 Shape 指针所指的對象不是 Circle 对象

或 Cylinder 对象的提示信息。函数 outputShapeArea 接收一个 Shape 指针作为它的参数,第一次调用收到 circle 的地址,第二次调用收到 cylinder 的地址,第 3 次调用收到基类 Shape 指针 ptr。

第 60 行

```
cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
```

用强制类型转换操作符 dynamic\_cast 动态执行向下类型转换,把 shapePtr(const Shape 类指针)转换为 const Cylinder 指针。cylinderPtr 将赋值为该 cylinder 对象的地址或 0。如果其值为 0 时,就表示 shapePtr 不是一个 cylinder 对象。如果类型转换结果不是 0,就输出该 cylinder 的面积。

第 67 行

```
circlePtr = dynamic_cast< const Circle * >( shapePtr );
```

用强制类型转换操作符 dynamic\_cast 动态执行向下类型转换把 shapePtr 为 const Circle \*。结果,circlePtr 将被赋值为该 Circle 对象的地址或者 0。如果它的值为 0,就表示 shapePtr 不是一个 Circle 对象。如果类型转换结果不是 0,就输出该 Circle 的面积。

常见编程错误 21.3 试图对 void 类指针使用 dynamic\_cast 是语法错误。

软件工程知识 21.4 RTTI 常与多重继承层次表(通过虚拟函数)同时使用。

## 21.8 操作符关键字

C++ 标准提供了一些操作符关键字(如图 21.8 所示),这些操作符关键字用于取代某些 C++ 操作符。如利用操作符关键字输入某些键盘不支持的一些特定字符如!,&^,~,| 等。

| 操作符        | 操作符关键字 | 描述     |
|------------|--------|--------|
| 逻辑操作符关键字   |        |        |
| &&         | and    | 逻辑 AND |
|            | or     | 逻辑 OR  |
| !          | not    | 逻辑 NOT |
| 不等操作符      |        |        |
| !=         | not_eq | 不等于    |
| 按位操作符关键字   |        |        |
| &          | bitand | 按位和    |
|            | bitor  | 按位或    |
| ^          | xor    | 按位异或   |
| ~          | compl  | 取反操作符  |
| 按位赋值操作符关键字 |        |        |
| &=         | and_eq | 按位和赋值  |
| =          | or_eq  | 按位或赋值  |
| ^=         | xor_eq | 按位异或赋值 |

图 21.8 用于取代操作符的操作符关键字

图 21.9 中的程序演示了操作符关键字的用法。该程序在 Microsoft Visual C++ 下编译通过,要使用操作符关键字,必须包含头文件 <iso646.h>。其他编译器可能会有所不同,查看

编译器文档,确定要包含的特定头文件(也许有些编译器不需包含任何头文件,即可使用这些操作符关键字)。

```

1 //Fig. 21.9: fig21_09.cpp
2 //Demonstrating operator keywords.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::boolalpha;
8
9 #include <iso646.h>
10
11 int main()
12 {
13     int a = 8, b = 22;
14
15     cout << boolalpha
16         << "    a and b: " << ( a and b )
17         << "\n    a or b: " << ( a or b )
18         << "\n    not a: " << ( not a )
19         << "\na not_eq b: " << ( a not_eq b )
20         << "\na bitand b: " << ( a bitand b )
21         << "\na bit_or b: " << ( a bitor b )
22         << "\n    a xor b: " << ( a xor b )
23         << "\n    compl a: " << ( compl a )
24         << "\na and_eq b: " << ( a and_eq b )
25         << "\na or_eq b: " << ( a or_eq b )
26         << "\na xor_eq b: " << ( a xor_eq b ) << endl;
27
28     return 0;
29 }
```

输出结果:

```

a and b: true
    a or b: true
    not a: false
a not_eq b: false
a bitand b: 22
a bit_or b: 22
    a xor b: 0
    compl a: -23
a and_eq b: 22
    a or_eq b: 30
a xor_eq b: 30
```

图 21.9 操作符关键字用法示例

程序中声明并初始化了两个整型变量 *a* 和 *b*,并使用了不同的操作符关键字执行 *a* 和 *b* 的逻辑和位操作,同时输出每一次操作的结果。

## 21.9 显式构造函数

第8章中,提到许多只带一个参数的构造函数都可供编译器完成隐式转换。而在这些隐式转换中,构造函数所收到的函数类型将转换为定义该构造函数的类的对象。这些隐式转换是自动进行的,不需要使用强制类型转换操作符。某些情况下,隐式转换并不适用且容易产生错误。例如,我们在图8.4中的Array类定义了一个只带有一个整型参数的构造函数。这个构造函数的意图是创建一个Array对象,这个Array对象将包含许多单元,而这些单元的个数将由整型参数指定。然而,该构造函数会被编译器错误地执行隐式转换。图21.10中的程序使用了取自第8章的Array类简单版本,演示错误的隐式转换。

```

1 //Fig 21.10: array2.h
2 //Simple class Array (for integers)
3 #ifndef ARRAY2_H
4 #define ARRAY2_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 class Array {
11     friend ostream &operator <<( ostream &, const Array & );
12 public:
13     Array( int = 10 ); //default /conversion constructor
14     ~Array();         //destructor
15 private:
16     int size; //size of the array
17     int *ptr; //pointer to first element of array
18 };
19
20 #endif

```

图 21.10 单参数构造函数和隐式转换——array2.h

```

21 //Fig 21.10: array2.cpp
22 //Member function definitions for class Array
23 #include <iostream>
24
25 using std::cout;
26 using std::ostream;
27
28 #include <cassert>
29 #include "array2.h"
30
31 //Default constructor for class Array (default size 10)
32 Array::Array( int arraySize )
33 {
34     size = ( arraySize > 0 ? arraySize : 10 );
35     cout << "Array constructor called for "

```

```

36         << size << " elements\n";
37
38     ptr = new int[ size ]; //create space for array
39     assert( ptr != 0 );    //terminate if memory not allocated
40
41     for ( int i = 0; i < size; i ++ )
42         ptr[ i ] = 0;      //initialize array
43 }
44
45 //Destructor for class Array
46 Array::~~Array() { delete [] ptr; }
47
48 //Overloaded output operator for class Array
49 ostream &operator <<( ostream &output, const Array &a )
50 {
51     int i;
52
53     for ( i = 0; i < a.size; i ++ )
54         output << a.ptr[ i ] << ' ';
55
56     return output;    //enables cout << x << y;
57 }

```

图 21.10 单参数构造函数和隐式转换——array2. cpp

```

58 //Fig 21.10: fig21_10.cpp
59 //Driver for simple class Array
60 #include <iostream>
61
62 using std::cout;
63
64 #include "array2.h"
65
66 void outputArray( const Array & );
67
68 int main()
69 {
70     Array integers1( 7 );
71
72     outputArray( integers1 );    //output Array integers1
73
74     outputArray( 15 );    //convert 15 to an Array and output
75
76     return 0;
77 }
78
79 void outputArray( const Array &arrayToOutput )
80 {
81     cout << "The array received contains:\n"
82         << arrayToOutput << "\n\n";
83 }

```

输出结果:

```
Array constructor called for 7 elements
The array received contains:
0 0 0 0 0 0 0

Array constructor called for 15 elements
The array received contains:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

图 21.10 单参数构造函数和隐式转换——fig21\_10.cpp

第 70 行的 main 中

```
Array integers1( 7 );
```

定义了 Array 类对象 `integers1`, 同时用整数 7 作为参数, 调用单参数构造函数, 指定 Array 内包含的单元个数。我们修改了 Array 构造函数以使它能输出一行文字, 这行文字表示调用 Array 构造函数以及分配到 Array 中的单元个数。第 72 行

```
outputArray( integers1 ); //output Array integers1
```

调用函数 `outputArray` (在第 79 行定义) 输出 Array 对象 `integers1` 中的内容。函数 `outputArray` 接收一个指向 Array 对象的 `const` 引用将其作为参数, 然后用重载的流插入操作符 `<<` 输出该 Array。第 74 行

```
outputArray( 15 ); //convert 15 to an Array and output
```

用整数 15 作为参数, 调用函数 `outputArray`。由于无任何一个函数 `outputArray` 带整型参数, 因此编译器查看 Array 类, 确定 Array 类中是否有转换构造函数能够把一个整型转换为 Array 对象。因为 Array 类提供了这样的转换构造函数, 所以编译器就用该构造函数创建了一个临时的 Array 对象, 这个对象包含了 15 个单元。最后, 编译器把这个临时的 Array 对象传给函数 `outputArray`, 输出这个临时的 Array 对象。输出表明调用了 Array 转换构造函数生成了一个含有 15 个单元的 Array 对象, 并且输出了这个 Array 对象的内容。

C++ 提供了关键字 `explicit`, 禁止通过转换构造函数进行隐式转换。声明为显式 (`explicit`) 类型的构造函数不能用于隐式转换。图 21.11 中的程序演示了显式构造函数的用法。

```
1 //Fig. 21.11: array3.h
2 //Simple class Array (for integers)
3 #ifndef ARRAY3_H
4 #define ARRAY3_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 class Array {
11     friend ostream &operator <<( ostream &, const Array & );
12 public:
13     explicit Array( int = 10 ); //default constructor
14     ~Array(); //destructor
15 private:
16     int size; //size of the array
17     int *ptr; //pointer to first element of array
18 };
```

```

19
20 #endif

```

图 21.11 显式构造函数用法示例——array3.h

```

21 //Fig. 21.11: array3.cpp
22 //Member function definitions for class Array
23 #include <iostream>
24
25 using std::cout;
26 using std::ostream;
27
28 #include <cassert>
29 #include "array3.h"
30
31 //Default constructor for class Array (default size 10)
32 Array::Array( int arraySize )
33 {
34     size = ( arraySize > 0 ? arraySize : 10 );
35     cout << "Array constructor called for "
36          << size << " elements\n";
37
38     ptr = new int[ size ]; //create space for array
39     assert( ptr != 0 );    //terminate if memory not allocated
40
41     for ( int i = 0; i < size; i ++ )
42         ptr[ i ] = 0;      //initialize array
43 }
44
45 //Destructor for class Array
46 Array::~~Array() { delete [ ] ptr; }
47
48 //Overloaded output operator for class Array
49 ostream& operator<<( ostream& output, const Array& a )
50 {
51     int i;
52
53     for ( i = 0; i < a.size; i ++ )
54         output << a.ptr[ i ] << ' ';
55
56     return output;    //enables cout << x << y;
57 }

```

图 21.11 显式构造函数用法示例——array3.cpp

```

58 //Fig. 21.11: fig21_11.cpp
59 //Driver for simple class Array
60 #include <iostream>
61
62 using std::cout;
63
64 #include "array3.h"
65

```



```

66 void outputArray( const Array & );
67
68 int main()
69 {
70     Array integers1( 7 );
71
72     outputArray( integers1 );    //output Array integers1
73
74     //ERROR; construction not allowed
75     outputArray( 15 );    //convert 15 to an Array and output
76
77     outputArray( Array( 15 ) ); //really want to do this!
78
79     return 0;
80 }
81
82 void outputArray( const Array &arrayToOutput )
83 {
84     cout << "The array received contains:\n"
85          << arrayToOutput << "\n\n";
86 }

```

Borland C++ 命令行编译器输出的错误消息:

Fig21\_11.cpp

Error E2064 Fig21\_11.cpp 18; Cannot initialize 'const Array &' with 'int' in function main ( )

Error E2340 Fig21\_11.cpp 18; Type mismatch in parameter 1 (wanted 'const Array &', got 'int') in function main ( )

\*\*\* 2 errors in Compile \*\*\*

Microsoft Visual C++ 编译器输出的错误消息:

Compiling...

Fig21\_11.cpp

Fig21\_11.cpp(18): error C2664: 'outputArray': cannot convert parameter 1 from 'const int' to 'const class Array &'

Reason: cannot convert from 'const int' to 'const class Array'

No constructor could take the source type, or constructor overload resolution was ambiguous

图 21.11 显式构造函数用法示例——fig21\_11.cpp

对于图 21.10 中的程序,所作的惟一修改在第 13 行,在单参数构造函数的声明前增加了关键字 `explicit`。编译这个程序时,编译器将产生一个错误消息。消息指出第 75 行传入函数 `outputArray` 的整数不能转换为 `const Array` 的引用。编译器的错误消息如输出窗口所示。第 77 行演示了如何用显式构造函数创建一个含有 15 个单元的 `Array` 类对象,以及如何把它传给函数 `outputArray`。

**常见编程错误 21.4** 试图调用显式构造函数执行隐式转换是语法错误。

**常见编程错误 21.5** 除用于单参数构造函数外,将关键字 `explicit` 用于数据成员和类成员函数是语法错误。

**软件工程知识 21.5** 对于不需要编译器进行隐式转换单参数构造函数,务必在该函数之间加上关键字 `explicit`。

## 21.10 mutable 类成员

21.4 小节介绍了操作符 `const_cast`,通过它我们可以强制转换常量特性。C++ 提供了存储类说明符 `mutable` 取代了操作符 `const_cast`。`mutable` 类数据成员永远都可以修改,即使是在常量成员函数或常量对象中。这样就就不必强制转换常量特性了。

**可移植性提示 21.2** 不管是用操作符 `const_cast` 还是用 C 风格的强制类型转换,试图改变常量对象的结果都将由编译器决定。

说明符 `mutable` 和操作符 `const_cast` 都可以改变数据成员,但两个关键字的使用情况不同。如果 `const` 对象没有使用 `mutable` 加以说明,那么每一次修改它都需要用 `const_cast` 操作符。这样极大减少了改变成员的偶然性,因为 `const` 对象的并不总是可修改的。涉及到操作符 `const_cast` 的操作通常隐藏在成员函数的实现中。类的用户可能不知道某个成员正在修改。

**软件工程知识 21.6** 当一个类有隐藏的实现细节,同时这些实现对其对象所包含逻辑值没有影响,这时,便非常适合用 `mutable` 类成员。

图 21.12 中的程序演示了如何使用 `mutable` 类成员。程序在第 8 行定义了一个 `TestMutable` 类,该类包含一个构造函数、两个成员函数和一个 `public mutable` 类数据成员 `value`。第 11 行

```
void modifyValue() const { value ++; }
```

定义了函数 `modifyValue`,同时把该函数定义为常量函数,在该函数为增加了 `mutable` 类数据成员 `value` 的值。通常情况下,我们不能改变数据成员,除非用 `const_cast` 操作符,把常量成员函数所操作的对象强制类型转换为成一个非常量对象,这里指的常量成员函数所操作的对象就是 `this` 指针所指向的对象。在该示例程序中,因为 `value` 成员变量是可变的,因此可以在常量函数内是修改它。第 12 行定义的函数 `getValue` 也是一个常量成员函数,它返回成员变量 `value` 的值。成员变量 `value` 也是可变的,所以也可以修改它。

```
1 //Fig. 21.12: fig21_12.cpp
2 //Demonstrating storage class specifier mutable.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class TestMutable {
9 public:
10     TestMutable( int v = 0 ) { value = v; }
11     void modifyValue() const { value ++; }
12     int getValue() const { return value; }
13 private;
```

```

14     mutable int value;
15 };
16
17 int main()
18 {
19     const TestMutable t( 99 );
20
21     cout << "Initial value; " << t.getValue();
22
23     t.modifyValue();    //modifies mutable member
24     cout << "\nModified value; " << t.getValue() << endl;
25
26     return 0;
27 }

```

输出结果:

```

Initial value: 99
Modified value: 100

```

图 21.12 mutable 数据成员用法示例

第 19 行声明了一个 const TestMutable 对象 t, 并把其初始化为 99。第 21 行输出了数据成员 value 的值。第 23 行调用的常量成员函数 modifyValue 使成员变量 value 增加 1。注意, 对象 t 和函数 modifyValue 都是常量类型。第 24 行输出了数据成员 value 的值(100), 这样即可证明了 mutable 数据成员已被修改。

## 21.11 类成员指针(. \* 和-> \* )

C++ 提供了访问类成员. \* 和-> \* 操作符。类成员指针与前文讨论的指针不同。试图通过类成员指针来使用-> 和 \* 操作符将产生语法错误。图 21.13 中的程序演示了类成员指针操作符的用法。

```

1 //Fig.21.13 fig21_13.cpp
2 //Demonstrating operators .* and -> *
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Test {
9 public:
10     void function() { cout << "function\n"; }
11     int value;
12 };
13
14 void arrowStar( Test * );
15 void dotStar( Test * );
16
17 int main()
18 {

```

```

19     Test t;
20
21     t.value = 8;
22     arrowStar( &t );
23     dotStar( &t );
24     return 0;
25 }
26
27 void arrowStar( Test *tPtr )
28 {
29     void ( Test::*memPtr )() = &Test::function;
30     ( tPtr -> *memPtr )(); //invoke function indirectly
31 }
32
33 void dotStar( Test *tPtr )
34 {
35     int Test::*vPtr = &Test::value;
36     cout << ( *tPtr ).*vPtr << endl; //access value
37 }

```

输出结果:

```

function
8

```

图 21.13 操作符 \* 和 -> \* 用法示例

**常见编程错误 21.6** 试图通过类成员指针来使用 -> 和 \* 操作符是语法错误。

上述程序声明了一个 Test 类,该类包含一个 public 成员函数 function 和一个 public 数据成员 value。函数 function 输出字符串“function”。第 14 和 15 行创建了函数 arrowStar 和函数 dotStar 的原型。第 19 ~ 21 行实例化了一个 Test 对象 t,并把 t 的数据成员 value 设置为 8。第 22 行和第 23 行调用了函数 arrowStar 和 dotStar,每次调用都传递了对象 t 的地址。

第 29 行

```
void ( Test::*memPtr )() = &Test::function;
```

在函数 arrowStar 中,声明了 memPtr 并将其初始化为指向类 Test 中成员的指针,而 Test 类的成员是一个函数,它是 void 类型不带参数。我们从上述赋值语句的左边开始研究。首先,void 是成员函数的返回类型;其次,空括号表明该成员函数不带任何参数;最后,中间的括号定义了一个指针 memPtr,该指针指向 Test 类的一个成员。Test:: \* memPtr 两侧的括号是必须的。如果不用 Test::,memPtr 就是一个标准函数指针。下面研究上述赋值语句右边的值。

**常见编程错误 21.7** 声明成员函数指针时,不用括号把该指针名字括起来是语法错误。

**常见编程错误 21.8** 声明成员函数指针时,不在该指针名字的前面加上类名和作用域分辨符是语法错误。

上述赋值语句右边用地址操作符(&)获得 function 函数的偏移量,function 函数应返回 void 且不带参数。指针 memPtr 初始化为这个偏移量。第 29 行赋值语句的左边和右边均针对任何特定对象。只有类名才与二元作用域分辨符(::)一起使用。如果不用 &Test::,第 29 行的赋值语句右边将是一个标准的函数指针。第 30 行

```
( pPtr -> * memPtr )();
```

用 `-> *` 操作符调用在指针 `memPtr` 内存储的成员函数(也就是 function)。第 35 行

```
int Test:: * vPtr = &Test::value;
```

声明了一个整型指针 `vPtr`,并将其初始化为指向类 `Test` 的整型数据成员。赋值语句右侧指定了数据成员 `value` 的偏移量。注意,如果不用 `Test::`,`vPtr` 就变成了一个指向整数 `value` 地址的整型指针。

下一条语句

```
cout << ( tPtr ).vPtr << endl;
```

用 `*` 操作符访问 `vPtr` 内存储的成员。注意,在客户代码中只能用成员指针操作符来访问成员。本例中,`value` 和 `function` 都是 `public` 类数据成员。在任何一个类成员函数内,类的所有成员都是可访问的。

**常见编程错误 21.9** 在 `.*` 和 `-> *` 这两个字符串中间加入空格是语法错误。

**常见编程错误 21.10** 改变字符在 `.*` 和 `-> *` 中的顺序是语法错误。

## 21.12 多重继承和虚拟基类

第 9 章,我们讨论了多重继承,即一个类可以由两个以上基类继承而来。比如,C++ 标准库就用多重继承来创建类 `iostream`(如图 21.14 所示)。

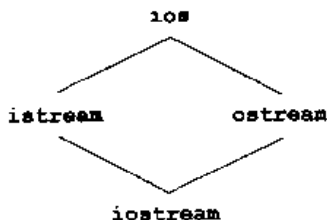


图 21.14 `iostream` 类的多重继承层次

`ostream` 类和 `istream` 类基类都是 `ios` 类,而且这两个类都是通过单一继承创建的。`iostream` 类从 `ostream` 类和 `istream` 类两个类继承而来。这样一来,`iostream` 类包含了 `ostream` 类和 `istream` 类这两个类所提供的所有功能。在多重继承层次中,图 21.14 所示的情况称为菱形继承。

因为 `ostream` 类和 `istream` 类均从 `ios` 类继承而来,所以 `iostream` 类可能存在问题。`iostream` 类将包含两个超类对象(因为 `ostream` 类和 `istream` 类都继承了 `ios` 类)。把 `iostream` 指针向上转换为 `ios` 指针时就会产生问题。这是因为存在两个 `ios` 子对象,而编译器不知道此时要使用那个子对象。这种歧义性会导致语法错误。图 21.15 演示了这种歧义性,只不过它是通过隐式转换来演示的而不是向下转换。当然,`iostream` 类不会出现这种问题。本节将介绍如何用虚拟基类来解决重复子对象问题。

```

1 //Fig. 21.15: fig21_15.cpp
2 //Attempting to polymorphically call a function
3 //multiply inherited from two base classes.
```

```

4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  class Base {
10 public:
11     virtual void print() const = 0; //pure virtual
12 };
13
14 class DerivedOne : public Base {
15 public:
16     //override print function
17     void print() const { cout << "DerivedOne\n"; }
18 };
19
20 class DerivedTwo : public Base {
21 public:
22     //override print function
23     void print() const { cout << "DerivedTwo\n"; }
24 };
25
26 class Multiple : public DerivedOne, public DerivedTwo {
27 public:
28     //qualify which version of function print
29     void print() const { DerivedTwo::print(); }
30 };
31
32 int main()
33 {
34     Multiple both; //instantiate Multiple object
35     DerivedOne one; //instantiate DerivedOne object
36     DerivedTwo two; //instantiate DerivedTwo object
37
38     Base *array[ 3 ];
39     array[ 0 ] = &both; //ERROR -- ambiguous
40     array[ 1 ] = &one;
41     array[ 2 ] = &two;
42
43     //polymorphically invoke print
44     for ( int k = 0; k < 3; k ++ )
45         array[ k ] -> print();
46
47     return 0;
48 }

```

Borland C++ 命令行编译器输出的错误消息:

Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland

Fig21\_15.cpp:

Error E2034 Fig21\_15.cpp 39: Cannot convert 'Multiple \*' to 'Base \*'  
in function main ( )

```
*** 1 errors in Compile ***
```

Microsoft Visual C++ 编译器输出的错误消息:

```
Compiling...
```

```
Fig21_15.cpp
```

```
fig21_15.cpp(39) : error c2594: '=' : ambiguous conversions from  
'class Multiple *' to 'class Base *'
```

图 21.15 多态调用多重继承函数

### 性能提示 21.1 重复子对象会耗费更多内存空间。

上述程序中,定义了 Base 类,它包含了纯虚拟函数 print。DerivedOne 类和 DerivedTwo 类以 public 方式继承了 Base 类,并且重载了函数 print。DerivedOne 类和 DerivedTwo 类都包含了一个 Base 类子对象。

Mutiple 类从 DerivedOne 类和 DerivedTwo 类多重继承而来。重载函数 print,调用 DerivedTwo 类中的函数 print。注意这里的限定条件,它指定了要调用子对象的哪个版本。

函数 main 中,为该继承层次中的每个类都创建了对象,并声明了一个 Base 类的指针数组。每个数组成员都初始化为对象的地址。当对象 both(属于多重继承类 Mutiple)的地址隐式转换为 Base 类指针时,将产生错误。对象 both 包含了从 Base 类继承而来的双重子对象,毫无疑问,这一点将使对函数 print 的调用产生歧义。程序中写了一个 for 循环,用该循环多态调用 array 数组中每个对象的 print 函数。

虚拟继承机制可以很好地解决重复子对象问题。基类被虚拟继承时,其派生类中只有一个子对象。我们称其为虚拟基类继承机制。图 21.16 中的程序修改了图 21.15 中的程序,演示了虚拟基类的用法。

```
1 //Fig.21.16: fig21_16.cpp
2 //Using virtual base classes.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Base {
9 public:
10     //implicit default constructor
11
12     virtual void print() const = 0; //pure virtual
13 };
14
15 class DerivedOne : virtual public Base {
16 public:
17     //implicit default constructor calls
18     //Base default constructor
19
20     //override print function
21     void print() const { cout << "DerivedOne\n"; }
```

```

22 |;
23
24 class DerivedTwo : virtual public Base {
25 public:
26     //implicit default constructor calls
27     //Base default constructor
28
29     //override print function
30     void print() const { cout << "DerivedTwo\n"; }
31 };
32
33 class Multiple : public DerivedOne, public DerivedTwo {
34 public:
35     //implicit default constructor calls
36     //DerivedOne and DerivedTwo default constructors
37
38     //qualify which version of function print
39     void print() const { DerivedTwo::print(); }
40 };
41
42 int main()
43 {
44     Multiple both;    //instantiate Multiple object
45     DerivedOne one;   //instantiate DerivedOne object
46     DerivedTwo two;   //instantiate DerivedTwo object
47
48     Base *array[ 3 ];
49     array[ 0 ] = &both;
50     array[ 1 ] = &one;
51     array[ 2 ] = &two;
52
53     //polymorphically invoke print
54     for ( int k = 0; k < 3; k ++ )
55         array[ k ] -> print();
56
57     return 0;
58 }

```

输出结果:

```

DerivedTwo
DerivedOne
DerivedTwo

```

图 21.16 虚拟基类用法示例

上述程序中,我们定义了 Base 类。其中包含了纯虚拟函数 print。DerivedOne 类从 Base 类继承而来,代码如下

```
class DerivedOne : virtual public Base {
```

同样地,DerivedTwo 类也是从 Base 类继承而来,代码如下

```
class DerivedTwo: virtual public Base {
```

两个类都从 Base 类继承而来,各自只包含一个 Base 子对象。Mutiple 类从 DerivedOne 类和



DerivedTwo 类继承而来。Mutiple 类只包含一个 Base 子对象。编译器现在允许执行这样的转换(从 Mutiple 指针类型转换为 Base 指针类型。在 main 函数中,为继承层次表中的每一个类创建了一个对象,同时声明了一个 Base 指针数组 array。数组 array 中的每一个成员都初始化为一个对象的地址。注意现在允许从 both 的地址到 Base 指针的向上转换。for 循环遍历了数组 array 中的每个成员,并多态调用这些对象的函数 print。

如果基类使用默认构造函数,虚拟基类继承层次的设计就很简单。前面两个例子使用编译器生成的默认构造函数。如果虚拟基类提供构造函数,设计就会变得很复杂,因为派生类最后必须初始化完成对虚拟基类。

以上两个例子中,Base, DerivedOne, DerivedTwo 和 Mutiple 均为最后派生类。创建 Base 对象时,Base 类是最后派生类;创建 DerivedOne(或 DerivedTwo)对象时,DerivedOne 类(或 DerivedTwo)是最后派生类;同样地,创建 Mutiple 对象时,Mutiple 类是最后派生类。无论类的继承层次有多深,都会有最后派生类,最后派生类必须负责初始化虚拟基类。在自测题 21.17 中,希望读者亲自实践最后派生类。

**软件工程知识 21.7** 为虚拟基类提供一个默认构造函数能简化继承层次表的设计。

## 21.13 结束语

衷心希望大家通过本书的学习,领略到学习 C++ 和面向对象编程的乐趣。前途无限光明,祝愿你在人生的道路上获得成功。欢迎提出意见、批评、纠正和建议。我们的邮件地址是:deitel@deitel.com 祝你好运!

## 21.14 小结

- C++ 标准提供了 bool 型数据类型(取值为 true 或 false)取代原有用 0 表示 false,用非 0 表示 true 的方式。
- 流操纵元 boolalpha 设置输出流把 bool 型变量的值显示为字符串“false”和“true”。
- C++ 标准介绍了 4 个强制类型转换操作符,这些操作符优于在 C 和 C++ 中使用的原始风格的强制类型转换操作符。
- C++ 提供了用于类型转换的 static\_cast 操作符。其中的类型检查在编译时完成。
- const\_cast 操作符转换对象的常量特性。
- reinterpret\_cast 操作符用于进行不相关类型之间的非标准类型转换。
- 每个名称空间都定义一个范围,其中可放置标识符和变量。要使用这些名称空间里面的成员,成员名字之前必须用名称空间的名字和二元作用域解析符限制,或者在用这些成员之前使用 using 语句。
- 名称空间可以包含常量、数据、类、嵌套名称空间以及函数等。名称空间必须定义在全局范围内或者嵌套于其他名称空间。
- 未命名名称空间的成员存在于全局名称空间。
- 运行值类型信息(RTTI)提供了一种方式,可在程序运行时确定对象的类型。

- 编译时,操作符 typeid 返回指向一个 type \_ info 的引用。type \_ info 对象是系统固有的对象,用于描述类型。
- 操作符 dynamic \_ cast 保证在运行时执行正常转换。执行非法类型转换操作时,dynamic \_ cast 返回 0。
- C++ 标准提供了操作符关键字,这些关键字可用于取代一些C++ 操作符。
- C++ 提供了关键字 explicit 限制通过转换构造函数进行的隐式转换。用 explicit 声明的构造函数不能用于进行隐式转换。
- mutable 数据成员是可修改的,即使是在 const 成员函数或者 const 对象。
- C++ 提供了访问类成员的. \* 和 -> \* 操作符。
- 多重继承将产生多个子对象,虚拟继承机制则能解决这个问题。基类被虚拟的继承时,其派生类中只有一个子对象(这个过程称为虚拟基类继承)。

## 本章术语

anonymous namespace 匿名名称空间

boolalpha 流操纵元

diamond inheritance 菱形继承

downcast 向下转换操作符

explicit conversion 显式转换

explicit 显式

global namespace 全局名称空间

global variables 全局变量

implicit conversion 隐式转换

most derived class 最后派生类

nested namespace 内嵌名称空间

ompl 位取反操作符

operator keywords 操作符关键字

pointer to class member operator 类成员指针操作符

pointer to data member 类成员变量指针

pointer to member function 类成员函数指针

RIIT(run - time type information) 运行时类型信息

subobject 子对象

virtual base class 虚拟基类

virtual 虚拟

## 常见编程错误

- 21.1 用操作符 static \_ cast 完成非法的类型转换会产生语法错误。这种非法类型转换包括:把常量转换为非常量;在没有公共继承关系的类型之间执行指针和引用的类型转换;转换一个类型,但这种类型没有可用于完成这类转换的相应构造函数或转换操作符。
- 21.2 把 main 关键字设置于名称空间内是语法错误。
- 21.3 试图对 void 型指针使用 dynamic \_ cast 是语法错误。
- 21.4 试图调用显式构造函数用于隐式转换是语法错误。
- 21.5 除了单参数构造函数,将关键字 explicit 用于据成员和类成员函数是语法错误。
- 21.6 试图通过指向类成员的指针来使用 -> 和 \* 操作符是语法错误。
- 21.7 声明成员函数指针时,不用括号把指针名字括起来,是语法错误。
- 21.8 声明成员函数指针时,不在该指针名字前加上类名和作用域分辨符,是语法错误。
- 21.9 在. \* 和 -> \* 这两个字符串中间放上空格将产生语法错误。
- 21.10 改变字符在. \* 和 -> \* 中的顺序将产生语法错误。

## 良好编程习惯

- 21.1 创建表明 true 和 false 值的状态变量时,用布尔值胜于用整数值。

- 21.2 为使程序更清晰,尽量用 true 和 false 代替零和非零。
- 21.3 尽量更安全、可靠的 static\_cast 操作符,避免用 C 类型的类型转换操作符。
- 21.4 避免用下划线( \_ )作为标识符的开头,以免出现链接错误。
- 21.5 为避免产生范围冲突,应尽量在成员前面加上它所在名称空间的名字和二元作用域分辨符(::)。
- 21.6 当一个类有隐藏的实现的细节,同时这些实现对其对象所包含逻辑值没有影响时,非常适合用 mutable 成员。

### 性能提示

- 21.1 重复子对象会耗费更多内存空间。

### 可移植性提示

- 21.1 在不同的平台用 reinterpret\_cast 可能使应用程序表现出不同的效果。
- 21.2 不管是操作符 const\_cast 还是 C 风格的类型转换,试图改变一个常量对象的结果都将由编译器而定。

### 软件工程知识

- 21.1 由于C++标准增加了新的强制类型转换操作符,因此C语言式的强制类型转换已经变得过时了。
- 21.2 每个编译单元都有自己惟一的未命名名称空间,也就是说未命名名称空间取代了 static 链接说明符。
- 21.3 最好在大型程序中把每个实体声明在一个类、函数、块或者名称空间内,这样能使每个实体的作用更突出。
- 21.4 RTTI 常与多重继承层次表(通过虚拟函数)同时使用。
- 21.5 对于不需要编译器进行隐式转换单参数构造函数务必在其之前使用 explicit 关键字。
- 21.6 当一个类有隐藏的实现细节,同时这些实现对其对象所包含逻辑值没有影响时,非常适合用 mutable 类成员。
- 21.7 为虚拟基类提供一个默认构造函数,可简化继承层次表的设计。

### 测试和调试提示

- 21.1 使用 reinterpret\_cast 时,比较容易执行一些危险操作,这些操作有可能引发严重的运行时错误。
- 21.2 为使用 RTTI,有些编译器需要启动 RTTI 功能。查看编译器的说明文档,了解RTTI的用法。

### 自测题

- 21.1 填空:
  - a) 操作符\_\_\_\_\_与名称空间一起使用以限定其成员。

b) 操作符\_\_\_\_\_转换对象的常量特性。

c) 操作符\_\_\_\_\_可用于转换类型。

21.2 判断正误。如果不正确,请说明原因。

a) 名称空间一定是惟一的。

b) 名称空间不能拥有一个名称空间作为其成员。

c) 布尔数据类型是一种基本的数据类型。

### 自测题答案

21.1 a) 二元作用域分辨符(::);

b) `const _ cast`;

c) C 风格的类型转换、`dynamic _ cast`、`static _ cast` 或者 `reinterpret _ cast`。

21.2 a) 不正确,程序员可以选择同名的名称空间。

b) 不正确,名称空间可以嵌套。

c) 正确。

### 练习题

21.3 填空题:

a) 操作符\_\_\_\_\_在运行时确定对象的类型。

b) 关键字\_\_\_\_\_指定使用的名称空间或者名称空间成员。

c) 操作符\_\_\_\_\_是逻辑 OR 的操作符关键字。

d) 存储说明符\_\_\_\_\_允许修改 `const` 成员修改。

21.4 判断正误。如不正确,请说明原因。

a) `static _ cast` 操作的合法性是在编译时检查。

b) `dynamic _ cast` 操作符的合法性是在运行时检查。

c) `typeid` 是一个关键字。

d) 关键字 `explicit` 能够用于构造函数、成员函数和数据成员。

21.5 指出下列表达式的求值结果(注意:有些表达式有错,请说明原因)。

a) `cout << false;`

b) `cout << ( bool b = 8 );`

c) `cout << ( a = true );` //a is of type int

d) `cout << ( *ptr + true && p );` // \*ptr is 10 and p is 8.88

e) // \* ptr is 0 and m is false

`bool k = ( *ptr * 2 || ( true + 24 ) );`

f) `bool s = true + false;`

g) `cout << boolalpha << false << setw( 3 ) << true;`

21.6 写一个 `namespace Currency`, 定义常量成员 `ONE`, `TWO`, `FIVE`, `TEN`, `TWENTY`, `FIFTY` 和 `HUNDRED`。写两个使用 `Currency` 的小程序。一个程序使用所有常量,另一个程序只使用。

21.7 写一个程序,用 `reinterpret _ cast` 操作符把不同的指针类型转换为整型。看是否有些转

换会发生语法错误?

- 21.8 写一个程序,用 `static_cast` 操作符把不同基本数据类型转换为整型。看编译器是否允许这种转换。
- 21.9 写一个程序,演示从派生类到基类的向上转换。用 `static_cast` 操作符完成这种向上转换。
- 21.10 写一个程序,生成一个 `explicit` 构造函数,取两个参数。编译器是否允许这样做? 删除关键字 `explicit` 并进行隐式转换,这时编译器是否允许这样做?
- 21.11 `explicit` 构造函数的好处何在?
- 21.12 写一个程序,创建一个类,令其包含两个构造函数。其中一个构造函数带一个整型的单参数,另一个取一个字符指针作为参数。另写一个程序,生成几个不同的对象,每个对象用不同的类型传入构造函数。不使用 `explicit` 关键字,会发生什么情况? 然后对带整型参数的构造函数使用 `explicit`,又会发生什么情况?
- 21.13 根据下面的 `namespace`,判断随后说法是否正确,如果有错,请说明原因。

```

1. #include <string>
2. namespace Misc {
3.     using namespace std;
4.     enum Countries { POLAND, SWITZERLAND, GERMANY,
5.                     AUSTRIA, CZECH_REPUBLIC };
6.     int kilometers;
7.     string s;
8.
9.     namespace Temp {
10.         short y = 77;
11.         Car car;    // assume definition exists
12.     }
13. }
14.
15. namespace ABC {
16.     using namespace Misc::Temp;
17.     void * function(void *,int );
18. }
```

- a) 可以在 `namespace ABC` 中访问变量 `y`。
  - b) 可以在 `namespace Temp` 中访问对象 `s`。
  - c) 不可在 `namespace Temp` 中访问常量 `POLAND`。
  - d) 可以在 `namespace ABC` 中访问常量 `GERMANY`。
  - e) 可以在 `namespace Temp` 中访问函数 `function`。
  - f) 可以在 `namespace Misc` 中访问名称空间 `ABC`。
  - g) 可以在 `namespace Misc` 中访问对象 `car`。
- 21.14 比较 `mutable` 和 `const_cast`。至少用一个例子说明一个比另一个好。注意:该练习题不要求编写任何代码。

- 21.15 写一个程序,用 `const_cast` 修改常量变量。(提示:用指针指向 `const` 标识符。)
- 21.16 虚拟基类能解决什么问题?
- 21.17 写一个程序,用虚拟基类。继承层次最顶层的类应提供至少取一个参数的构造函数 (而不是提供默认构造函数)。这种情况下的继承层次会出现什么问题?
- 21.18 指出下列各题中的错误,并说明如何改正。

- a) 

```
namespace Name {  
    int x,y;  
    mutable int x;  
};
```
- b) 

```
int integer = const_cast<int>(double);
```
- c) 

```
namespace PCM(111, "hello"); //construct namespace
```
- d) 

```
explicit int x = 99;
```

## 附录 A 操作符的优先级和结合性

操作符按优先级从高到低降序排列。

| 操作符                     | 类型                   | 结合性  |
|-------------------------|----------------------|------|
| ::                      | 二元作用域分辨符             | 从左到右 |
| :::                     | 一元作用域分辨符             |      |
| ( )                     | 括号                   | 从左到右 |
| [ ]                     | 数组下标                 |      |
| .                       | 通过对象的成员选定            |      |
| ->                      | 通过指针的成员选定            |      |
| ++                      | 一元后自增                |      |
| --                      | 一元后自减                |      |
| typeid                  | 运行时类型信息              |      |
| dynamic_cast < 类型 >     | 运行时经类型检查的强制类型转换      |      |
| static_cast < 类型 >      | 编译时经类型检查的强制类型转换      |      |
| reinterpret_cast < 类型 > | 非标强制类型转换             |      |
| const_cast < 类型 >       | 通过强制类型转换,除去 const 属性 |      |
| ++                      | 一元前自增                | 从右到左 |
| --                      | 一元前自减                |      |
| +                       | 一元加                  |      |
| -                       | 一元减                  |      |
| !                       | 一元逻辑否定               |      |
| ~                       | 一元按位求模               |      |
| ( 类型 )                  | C 风格的一元强制类型转换        |      |
| sizeof                  | 以字节为单位判断大小           |      |
| &                       | 地址                   |      |
| *                       | 间接引用                 |      |
| new                     | 动态内存分配               |      |
| new[ ]                  | 动态数组分配               |      |
| delete                  | 动态内存解除分配             |      |
| delete[ ]               | 动态数组解除分配             |      |
| . *                     | 通过对象的成员指针            | 从左到右 |
| -> *                    | 通过指针的成员指针            |      |
| *                       | 乘                    | 从左到右 |
| /                       | 除                    |      |
| %                       | 求模                   |      |
| +                       | 加                    | 从左到右 |
| -                       | 减                    |      |
| <<                      | 按位向左移位               | 从左到右 |
| >>                      | 按位向右移位               |      |
| <                       | 关系小于                 | 从左到右 |
| <=                      | 关系小于或等于              |      |

(续表)

| 操作符 | 类型       | 结合性  |
|-----|----------|------|
| >   | 关系大于     |      |
| >=  | 关系大于或等于  |      |
| =   | 关系等于     | 从左到右 |
| !=  | 关系不等于    |      |
| &   | 按位与      | 从左到右 |
| ^   | 按位异或     | 从左到右 |
|     | 按位或      | 从左到右 |
| &&  | 逻辑与      | 从左到右 |
|     | 逻辑或      | 从左到右 |
| ?:  | 三元条件     | 从右到左 |
| =   | 赋值       | 从右到左 |
| +=  | 加后赋值     |      |
| -=  | 减后赋值     |      |
| *=  | 乘后赋值     |      |
| /=  | 除后赋值     |      |
| %=  | 求模后赋值    |      |
| &=  | 按位与后赋值   |      |
| ^=  | 按位异或后赋值  |      |
| =   | 按位或后赋值   |      |
| <<= | 按位左移位后赋值 |      |
| >>= | 按位右移位后赋值 |      |
| ,   | 逗号       |      |

图 A.1 操作符的优先级和结合性



## 附录 B ASCII 字符集

| 0  | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |     |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | nul | soh | stx | etx | eot | enq | ack | bel | bs  | ht  |
| 1  | nl  | Vt  | ff  | cr  | So  | si  | dle | dc1 | dc2 | dc3 |
| 2  | dc4 | nak | syn | etb | can | em  | sub | esc | fs  | gs  |
| 3  | rs  | us  | sp  | !   | "   | #   | \$  | %   | &   | '   |
| 4  | (   | )   | *   | +   | ,   | -   | .   | /   | 0   | 1   |
| 5  | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | :   | ;   |
| 6  | <   | =   | >   | ?   | @   | A   | B   | C   | D   | E   |
| 7  | F   | G   | H   | I   | J   | K   | L   | M   | N   | O   |
| 8  | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   |
| 9  | Z   | [   | \   | ]   | ^   | _   | `   | a   | b   | c   |
| 10 | d   | e   | f   | g   | h   | i   | j   | k   | l   | m   |
| 11 | n   | o   | p   | q   | r   | s   | t   | u   | v   | w   |
| 12 | x   | y   | z   | {   |     | }   | ~   | del |     |     |

图 B.1 ASCII 字符集

表左边的数字是与对应的十进制值字符码(0 ~ 127)的高位,表顶部的数字则是字符码的低位。例如,“F”的字符码是 70,“&”的字符码则是 38。

# 附录 C 数值系统

## 学习目标

- 理解基本的数值系统概念,如基数、位置值和符号值
- 理解如何用二进制、八进制和十六进制表示数值
- 能将二进制数简写成八进制或十六进制数值
- 能将八进制和十六进制数字转换成二进制数值
- 会进行十进制、二进制、八进制和十六进制之间的互相转换
- 掌握二进制算术,以及如何用补值记号法表示负数

## C.1 简介

本附录,我们打算介绍程序员经常遇到的一些重要数值系统。特别是他们在开发需要同“机器级”硬件直接打交道的软件项目时。像这样的软件项目主要包括:操作系统、计算机网络软件、编译程序、数据库系统以及一些需要发挥最高性能的应用程序。

我们在程序中写入 227 或 -63 之类的整数时,已经假定了这些数值用十进制数值系统(或以 10 为基数)来表示。在十进制系统中,我们可用的“数位”包括 0,1,2,3,4,5,6,7,8 和 9。注意最小的数位是 0,而最大的数位是 9——比基数 10 小 1。不过,计算机内部使用的却是“二进制数值系统”以 2 为基数。在这种系统中,可用的数位只有两个:0 和 1。注意最小的数位是 0,最大的数位是 1(同样比基数 2 小 1)。图 C.1 总结了二进制、八进制、十进制以及十六进制数值系统中所用的数位。

| 二进制数位 | 八进制数位 | 十进制数位 | 十六进制数位       |
|-------|-------|-------|--------------|
| 0     | 0     | 0     | 0            |
| 1     | 1     | 1     | 1            |
|       | 2     | 2     | 2            |
|       | 3     | 3     | 3            |
|       | 4     | 4     | 4            |
|       | 5     | 5     | 5            |
|       | 6     | 6     | 6            |
|       | 7     | 7     | 7            |
|       |       | 8     | 8            |
|       |       | 9     | 9            |
|       |       |       | A (十进制值是 10) |
|       |       |       | B (十进制值是 11) |
|       |       |       | C (十进制值是 12) |
|       |       |       | D (十进制值是 13) |
|       |       |       | E (十进制值是 14) |
|       |       |       | F (十进制值是 15) |

图 C.1 二进制、八进制、十进制以及十六进制数值系统所用的数位

大家可以看到,同样的数字,其二进制值显然比其十进制值长出许多。汇编语言和高级语言虽然能深入到“机器级”实现编程,但对于使用这些语言程序员来说,二进制数的使用无疑当麻烦。于是又出现了另两种数值系统(八进制数值系统和十六进制数值系统,前者基数为8,后者基数为16)。这两种数值系统因为能方便地简化二进制数而大受欢迎。

八进制系统的数位范围为0~7。由于二进制和八进制数值系统采用的数位都比十进制系统少,所以两者不必重新设计新的数位而是“借用”十进制系统的数位写法。

十六进制则不同,它比十进制多6个数位——最小的数位仍然是0,但最大的数位相当于十进制的15(比基数16小1)。因此,我们按照习惯用A~F这6个字母代表多出的那几个十六进制数位,分别对应于十进制的10~15。所以在十六进制中,我们看到多种形式的数字:比如完全由十进制的数位组成的876(尽管它本身是十六进制数),同时使用了数位和字母的8A55F(显然也是十六进制数),则完全由字母组成的FFE(同样是十六进制数)。有时,程序员甚至能在十六进制数中发现看到FACE或FEED之类的有意义的单词。这对习惯于处理数字的程序员来说,可能会觉得奇怪。在图C.2中,我们简要总结了各种数值系统的基数和数位。

| 属性   | 二进制 | 八进制 | 十进制 | 十六进制 |
|------|-----|-----|-----|------|
| 基数   | 2   | 8   | 10  | 16   |
| 最小数位 | 0   | 0   | 0   | 0    |
| 最大数位 | 1   | 7   | 9   | F    |

图 C.2 二进制、八进制、十进制以及十六进制系统基数/数位

每种数值系统都采用了“按位记数法”,即每个数位都有一个不同的“位值”。举个例子来说,在十进制数937中(注意,9,3和7叫作“符号值”),我们认为7在“个位”,3在“十位”,而9在“百位”。注意每个位值是“基数”(基数是10)的乘幂,而且这些乘幂从0开始,依次向左自增1。图C.3对此进行了清楚的说明。

| 十进制数值系统的位值    |        |        |        |
|---------------|--------|--------|--------|
| 十进制数位         | 9      | 3      | 7      |
| 位表名           | 百      | 十      | 个      |
| 位表值           | 100    | 10     | 1      |
| 位表值是基数(10)的乘幂 | $10^2$ | $10^1$ | $10^0$ |

图 C.3 十进制系统的位置值设定

较长的十进制数,百位左侧的位值依次为千位( $10^3$ ),万位( $10^4$ ),十万位( $10^5$ ),百万位( $10^6$ ),千万位( $10^7$ )等。

类似地,在二进制数101中,我们认为最右边的1是“一位”,0是“二位”,最左边的1是“四位”。注意每个位值必然是“基数”(二进制的基数是2)的乘幂,而且这些乘幂从0次幂开始,依次向左自增1,比如 $2^0, 2^1, 2^2$ 等等。图C.4对此进行了总结。

对更长的二进制数,沿左侧的位值依次为8位( $2^3$ ),16位( $2^4$ ),32位( $2^5$ ),64位( $2^6$ )等。

同样地,在八进制数425中,我们认为5在“个位”,2在“八位”,而4在“六十四位”。注

意每个位值必然是“基数”(八进制的基数为8)的乘幂,而且这位值要从0次幂开始,以后每次向左自增1,比如 $8^0, 8^1, 8^2$ 等等。图 C.5 对此进行了总结。

| 二进制系统的位值    |       |       |       |
|-------------|-------|-------|-------|
| 二进制数位       | 1     | 0     | 1     |
| 位名          | 4     | 2     | 个     |
| 位值          | 4     | 2     | 1     |
| 位值是基数(2)的乘方 | $2^2$ | $2^1$ | $2^0$ |

图 C.4 二进制系统的位值设定

| 八进制系统的位值    |       |       |       |
|-------------|-------|-------|-------|
| 八进制数位       | 4     | 2     | 5     |
| 位名          | 64    | 8     | 个     |
| 位值          | 64    | 8     | 1     |
| 位值是基数(8)的乘幂 | $8^2$ | $8^1$ | $8^0$ |

图 C.5 八进制系统的位置值设定

| 十六进制数值系统的位置值 |        |        |        |
|--------------|--------|--------|--------|
| 十六进制数位       | 3      | D      | A      |
| 位名           | 256    | 16     | 个      |
| 位值           | 256    | 16     | 1      |
| 位值是基数(16)的幂  | $16^2$ | $16^1$ | $16^0$ |

图 C.6 十六进制系统的位值设定

对于更长的八进制数字,沿左侧的位值为512位( $8^3$ )、4 096位( $8^4$ )、32 768位( $8^5$ )等等。

最后,在十六进制数3DA中,A在个位,D在16位,而3在256位。注意每个位值都是“基数”(十六进制的基数是16)的一个乘幂,而且这些位值要从0次幂开始,以后每次向左自增1次方,比如 $16^0, 16^1, 16^2$ 等等。图 C.6 对此进行了总结。

对于更长的十六进制数,沿左侧的下一个位置值依次为4 096位( $16^3$ )、65 536位( $16^4$ )等。

## C.2 将二进制数简化为八进制和十六进制数

在计算领域,八进制和十六进制数的主要用途是把过长的二进制数字简化为更精简的形式。图 C.7 清楚地揭示了如何将冗长的二进制数字简洁地转换为基数更高的数值系统表示形式。

比较八进制/十六进制系统与二进制系统,我们可总结出一个重要的规律:八和十六进制系统的“基数”(8和16)均为二进制系统“基数”(2)的整数次幂。观察下而这个共包含了12个数位的二进制数,我们在它旁边列出了其相应的八进制和十六进制数。你能总结出一种最有效的方式,把二进制数迅速简写为八或十六进制吗?

| 十进制 | 二进制   | 八进制 | 十六进制 |
|-----|-------|-----|------|
| 0   | 0     | 0   | 0    |
| 1   | 1     | 1   | 1    |
| 2   | 10    | 2   | 2    |
| 3   | 11    | 3   | 3    |
| 4   | 100   | 4   | 4    |
| 5   | 101   | 5   | 5    |
| 6   | 110   | 6   | 6    |
| 7   | 111   | 7   | 7    |
| 8   | 1000  | 10  | 8    |
| 9   | 1001  | 11  | 9    |
| 10  | 1010  | 12  | A    |
| 11  | 1011  | 13  | B    |
| 12  | 1100  | 14  | C    |
| 13  | 1101  | 15  | D    |
| 14  | 1110  | 16  | E    |
| 15  | 1111  | 17  | F    |
| 16  | 10000 | 20  | 10   |

图 C.7 等值的十进制、二进制、八进制和十六进制数

|              |      |      |
|--------------|------|------|
| 二进制          | 八进制  | 十六进制 |
| 100011010001 | 4321 | 8D1  |

事实上,只需将该二进制数中的 12 个数位分为 4 组,每组包含 3 个数位,即可看出它和八进制的对应关系。如下所示

|     |     |     |     |
|-----|-----|-----|-----|
| 100 | 011 | 010 | 001 |
| 4   | 3   | 2   | 1   |

显然,在每组二进制数位之下,转换后的八进制数位和图 C.7 总结的对应关系完全一致。

同样的关系也适用于把二进制数字转换为十六进制。不过,这一次需要把该二进制数的 12 个数位分为 3 组,每组包含 4 个数位。如下所示

|      |      |      |
|------|------|------|
| 1000 | 1101 | 0001 |
| 8    | D    | 1    |

显然,在每组二进制数位之下,转换后的十六进制数位和图 C.7 总结的对应关系完全一致。

### C.3 将八进制和十六进制数转换为二进制数

上一节,讲述了如何将二进制数转换为八和十六进制数,具体的做法是对二进制数位进行分组,找出与每一组数位对应的八或十六进制形式,再合并结果即可。这一操作过程实际是“可逆”的,利用它可将指定的八或十六进制数方便地转换成二进制数。

以八进制数字 653 为例,可把 6 写为 3 位二进制形式,即 110;把 5 写为 3 位二进制形式,即 101;再把 3 写为 3 位二进制形式,即 011。最终得到一个共有 9 位的二进制数:110 101 011。

十六进制到二进制的转换也不例外。以 FAD5 为例,可把 F 写为 4 位二进制形式,即

1111;把 A 写为 1 010;把 D 写为 1 101;再把 5 写为 0 101。最终结果是:1111101011010101。

## C.4 将二进制、八进制或十六进制转换为十进制

我们平常习惯于使用十进制,所以似乎只有将二进制、八进制或十六进制数转换十进制之后,才能真正了解数值的含义。事实上,C.1 节讨论的位值中,我们使用的全是“十进制”。要想把其他进制的数转换到十进制,可以先让每个数位乘以其“位值”,再把这些结果加到一起,得到一个十进制数。二进制数 110 101 为例,转换后得到十进制数 53,如图 C.8 所示。

| 把二进制数转换为十进制 |                                  |               |             |             |             |             |
|-------------|----------------------------------|---------------|-------------|-------------|-------------|-------------|
| 位值          | 32                               | 16            | 8           | 4           | 2           | 1           |
| 符号值         | 1                                | 1             | 0           | 1           | 0           | 1           |
| 乘积          | $1 * 32 = 32$                    | $1 * 16 = 16$ | $0 * 8 = 0$ | $1 * 4 = 4$ | $0 * 2 = 0$ | $1 * 1 = 1$ |
| 和           | $= 32 + 16 + 0 + 4 + 0 + 1 = 53$ |               |             |             |             |             |

图 C.8 把二进制数转换为十进制

为了把八进制数 7 614 转换为十进制,我们采用类似于上文的操作,不过这一次换为相应的八进制位值。转换后的十进制数 3 980,如图 C.9 所示。

| 把八进制数转换为十进制 |                               |                |             |             |
|-------------|-------------------------------|----------------|-------------|-------------|
| 位值          | 512                           | 64             | 8           | 1           |
| 符号值         | 7                             | 6              | 1           | 4           |
| 乘积          | $7 * 512 = 3584$              | $6 * 64 = 384$ | $1 * 8 = 8$ | $4 * 1 = 4$ |
| 和           | $= 3584 + 384 + 8 + 4 = 3980$ |                |             |             |

图 C.9 把八进制数转换为十进制

同样地,我们用类似的办法把十六进制数 AD3B 转换为十进制,只不过这次换用了十六进制的位值设定。最终的十进制结果为 44 347 如图 C.10 所示。

| 把十六进制数转换为十进制 |                                    |                  |               |              |
|--------------|------------------------------------|------------------|---------------|--------------|
| 位值           | 4096                               | 256              | 16            | 1            |
| 符号值          | A                                  | D                | 3             | B            |
| 乘积           | $A * 4096 = 40960$                 | $D * 256 = 3328$ | $3 * 16 = 48$ | $B * 1 = 11$ |
| 和            | $= 40960 + 3328 + 48 + 11 = 44347$ |                  |               |              |

图 C.10 把十六进制数转换为十进制

## C.5 将十进制转换为二进制、八进制或十六进制

上一节,我们用固定不变的“位值”来实现数值从其他进制向十进制的转换。同样地,我们也可利用这一技术把十进制数转换成二、八以及十六进制。

现在,假定我们想把十进制数 57 转换为二进制。首先,请按从右到左的顺序,依次列出

二进制系统的各个“位值”，直到遇到某个位值大于我们要转换的那个十进制数为止。比如对 57 来说，我们可记为

位值:64 32 16 8 4 2 1

由于 64 已经大于 57，所以停止记录。然后从中删去大于 57 的 64，结果如下

位值:32 16 8 4 2 1

接着，按从左到右的顺序，依次处理上述列表中的值。首先用 32 除 57，得到商为 1，余数为 25，所以在“32”那一列的下方记下 1；再用 16 除上一次的余数 25，得到商为 1，余数为 9，所以在“16”那一列的下方记下 1；再用 8 除上一次的余数 9，得到商为 1，余数为 1，所以在 8 那一列的下方也记下“1”；用同样的方法对后续两列进行运算，得到的商都为 0，余数都为 1，所以在“4”和“2”这两列的下方分别记下 0；最后，1 除 1 的商为 1，所以在“1”这一列的下方要记下 1。结果如下

位值:32 16 8 4 2 1

符号值:1 1 1 0 0 1

因此，十进制数 57 等于二进制数 111 001。

类似地，要想把十进制数 103 转换成八进制，同样按从右到左的顺序，依次写下八进制的各个位值，直到遇到某个位值大于我们要转换的那个十进制数为止。因此，我们首先写下：

位值:512 64 8 1

随后删去那个大于 103 的位值 512，变为

位值:64 8 1

接着，从最左边的一列开始，向右依次处理每个数位。首先用 64 来除 103，得到商为 1，余数为 39，所以在“64”那一列的下方记下 1；再用 8 来除上一次的余数 39，得到商为 4，余数为 7，所以在“8”那一列的下方记下 4；最后，用 1 来除上一次的余数 7，得到商为 7，余数为 0，因为在“1”那一列的下方记下 7。结果如下

位值:64 8 1

符号值:1 4 7

因此，十进制数 103 等于八进制数 147。

同样地，要想把十进制数字 375 转换为十六进制，首先记下十六进制的各个位值，然后同样按从右到左的顺序，直到遇到一个位值大于要转换的十进制数字为止。因此，首先记下

位值:4096 256 16 1

由于 4096 已经大于要转换的 375，所以弃之，变为

位值:256 16 1

接下来的操作和以前相似。从最左边的那一列开始，依次向右对每一列进行处理。首先用 256 来除 375，得到商为 1，余数为 119，所以在“256”那一列记下 1；再用 16 来除上一次的余数 119，得到商为 7，余数也为 7，所以在“16”那一列记下 7；最后用 1 来除上一次的余数 7，得到商为 7，余数为 0，所以在“1”列下方记下 7。结果如下

位值:256 16 1

符号值:1 7 7

换言之，十进制数 375 等于十六进制数 177。

## C.6 负的二进制数:2 的补值记号法

迄今为止,本附录讨论的都是如何对正数进行计算与转换。在本节,我们打算解释一下计算机如何利用 2 的补值记号法来表示负数。对二进制数而言,首先需要掌握 2 的补值记号法是如何构成的。之后才可顺利理解为什么可以用它表示负的二进制数。

以目前最流行的、支持 32 位整数的计算机为例。假设

```
intvalue = 13;
```

那么把 value 换算为 32 位二进制位后,可得到结果

```
00000000 00000000 00000000 00001101
```

要想表示 value 的负值,首先需要采用 C++ 的按位取反操作符 (~),得到它的“1 的补值”结果。如下所示

```
onesComplementOfValue = ~ value
```

在内部,~ value 现在变成了 value 的每个二进制位取反后的结果:1 变为 0,而 0 变为 1。如下所示

```
value;
00000000 00000000 00000000 00001101
~ value(即 value 的 1 补值计算结果):
11111111 11111111 11111111 11110010
```

好了,为了得到 value 的“2 的补值”结果,只需在 ~ value 的基础上加 1 即可。结果如下

```
value 的 2 补值计算结果:
11111111 11111111 11111111 11110011
```

现在,假如上述值相当于 -13,那么再加上 13 的二进制值之后,应得到的结果为 0。不妨试验一下

```

00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----
00000000 00000000 00000000 00000000
```

显然,我们的推论是正确的,结果真的为 0! 假如把一个数的 1 的补值加到这个数字,结果数全是 1。之所以会得到全是 0 的结果,关键在于 2 的补值要比 1 的补值大 1。由于加了一个 1,所以每一列都加了 0,同时进位为 1。由此不断向左边进位,直至最左边的二进制位,造成结果数位全部变成 0(加 0 不进位,加 1 进位)。

执行减法运算

```
x = a - value;
```

时,实际是以如下所示的形式

```
$ x = $ a + ( ~ $ value + 1 );
```

把 value 的 2 的补值加到 a 上。假定 a 等于 27,而 value 等于 13,那么对 value 求 2 的补值时,实际相当于求它的负值。最后把这个值与 a 相加,结果为 14。可以亲自实践一下

```
a(等于 27)          00000000 00000000 00000000 00011011
```



```

+ (~value + 1)      +11111111 11111111 11111111 11110011
                     -----
                     00000000 00000000 00000000 00001110

```

结果真的等于 14!

## C.7 小结

- 在程序中写下 19,227 或 -63 这样的整数时,实际已假定它采用的是十进制(基数为 10)数值系统。在十进制数系统中,可用的数位包括 0,1,2,3,4,5,6,7,8 和 9。最小的数位是 0,最大的是 9(比基数 10 小 1)。
- 在计算机内部,必须采用二进制(基数为 2)数值系统。二进制数值系统只用到了两个数位:0 和 1。最小的数位是 0,最大的数位是 1(同样比基数 2 小 1)。
- 作为数值系统,八进制数值系统(基数为 8)和十六进制数值系统(基数为 16)比二进制数值系统更流行,因为它们可以方便地简化过于冗长的二进制数。
- 八进制数值系统可用的数位范围在 0~7 之间。
- 十六进制数值系统为我们引入了一个新问题,因为它要求用 16 个数位:最小数位仍然是 0,但最大数位相当于十进制的 15(同样比基数 16 小 1)。按照习惯,我们用 A~F 的 6 个字母来表示与十进制 10~15 对应的 6 个十六进制数位。
- 每种数值系统都有自己的一套“位置记号法”:每个数位所在的位置都有一个不同的“位值”。
- 八、十六进制数值系统与二进制数值系统的一个重要关系在于:八和十六进制系统的“基数”(分别为 8 和 16)均为二进制系统的“基数”(2)的整数幂。
- 为了把八进制数转换成二进制,只需将每个八进制数位转换为 3 个二进制数位,再把各组数位合并到一起,得到最终的二进制数。
- 要想把十六进制数转换成二进制,只需将每个十六进制数位转换成 4 个二进制数位,再把各组数位合并到一起,便得到了最终的二进制数。
- 由于我们平常已习惯了使用十进制,所以只有把二进制、八进制或十六进制数换算成十进制后,才会了解这些数字的真正含义。
- 为了把数字从其他数值表示法转换为十进制,只需让每个数位来去乘以它的位值,再把所有乘积加起来,便得到了最终的十进制结果。
- 计算机用 2 的补值记号法来表示负数。
- 要想用二进制数表示负数,首先要求出 1 的补值运算结果,这是用 ~ 操作符来实现的。结果会造成一个值的所有二进制位取值,所有 1 变为 0,所有 0 变为 1。在此基础上,求出 2 的补值运算结果,具体的做法很简单,只需在 1 的补值结果上加 1。这样一来,我们最终得到的便是一个负数(反数)。

### 术语

base 基数

base 2 number system 基数为 2 的数值系统

base 8 number system 基数为 8 的数值系统

base 10 number system 基数为 10 的数值系统

base 16 number system 基数为 16 的数值系统  
 binary number system 二进制数值系统  
 bitwise complement operator ( ~ )  
     按位取反操作符 ( ~ )  
 conversions 转换  
 decimal number system 十进制数值系统  
 digit 数位  
 hexadecimal number system 十六进制数值系统

negative value 负数  
 octal number system 八进制数值系统  
 one's complement notation 1 的补值记号法  
 positional notation 位置记号  
 positional value 位值  
 symbol value 符号值  
 two's complement notation 2 的补值记号法

## 自测题

- C.1 十进制、二进制、八进制和十六进制数值系统的“基数”分别为\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
- C.2 (选择题)通常,分别用十进制、八进制和十六进制来表达同一个数字时,会比其二进制表达形式(长/短)得多。
- C.3 (判断正误)大家之所以更喜欢用十进制,是由于它能方便地简化冗长的二进制数字。只需划分出 4 位一组的二进制数位,再把每一组转换成对应的十进制数位即可。
- C.4 对于较大的二进制数时,用八进制/十六进制/十进制可取得精简的结果。
- C.5 (判断正误)任何进制表示的数值系统,可用的最大数位肯定比其基数本身大 1。
- C.6 (判断正误)任何进制表示的数值系统,可用的最小数位肯定要比其基数本身小 1。
- C.7 在二进制、八进制、十进制或十六进制中,数字最右边那个数位的位值肯定是\_\_\_\_\_。
- C.8 在二进制、八进制、十进制或十六进制中,右边倒数第二个数位的位值肯定等于\_\_\_\_\_。
- C.9 请补全下面各数值系统右边四个数位的位值:

|      |      |     |     |     |
|------|------|-----|-----|-----|
| 十进制  | 1000 | 100 | 10  | 1   |
| 十六进制 | ...  | 256 | ... | ... |
| 二进制  | ...  | ... | ... | ... |
| 八进制  | 512  | 64  | 8   | 1   |

- C.10 把二进制数 110101011000 分别转换为八进制和十六进制。
- C.11 把十六进制数 FACE 转换为二进制。
- C.12 把八进制数 7316 转换为二进制。
- C.13 把十六进制数 4FEC 转换为八进制(提示:首先转换为二进制,再由二进制转换为八进制)。
- C.14 把二进制数 1101110 转换为十进制。
- C.15 把八进制数 317 转换为十进制。
- C.16 把十六进制数 EFD4 转换为十进制。
- C.17 把十进制数 177 分别转换为二进制、八进制和十六进制。
- C.18 最后列出十进制数 417 的二进制形式,然后列出 417 的 1 的补值计算结果,最后列出 417 的 2 的补值计算结果。
- C.19 一个值加上它的 1 的补值,会得到什么结果?

## 自测题答案

- C.1 10,2,8,16  
 C.2 短。  
 C.3 错误。  
 C.4 十六进制。  
 C.5 错误。任何进制的最大数位肯定要比基数小1。  
 C.6 错误。任何进制的最小数位肯定是0。  
 C.7 1(基数的0次幂)。  
 C.8 数值系统的基数。  
 C.9 补全之后的结果如下:

|      |      |     |    |   |
|------|------|-----|----|---|
| 十进制  | 1000 | 100 | 10 | 1 |
| 十六进制 | 4096 | 256 | 16 | 1 |
| 二进制  | 8    | 4   | 2  | 1 |
| 八进制  | 512  | 64  | 8  | 1 |

- C.10 八进制 6530;十六进制 D58  
 C.11 二进制 1111101011001110  
 C.12 二进制 111011001110  
 C.13 先转换为二进制:010011111101100;再转换为八进制:47754  
 C.14 十进制  $2 + 4 + 8 + 32 + 64 = 110$   
 C.15 十进制  $7 + 1 * 8 + 3 * 64 = 7 + 8 + 192 = 207$   
 C.16 十进制  $4 + 13 * 16 + 15 * 256 + 14 * 4096 = 61396$   
 C.17 十进制 177 转换为二进制后的结果:

```

256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001

```

转换为八进制后的结果:

```

512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261

```

转换为十六进制后的结果:

```

256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1

```

- C.18 二进制:

```

512 256 128 64 32 16 8 4 2 1

```

256 128 64 32 16 8 4 2 1  
 $(1 * 256) + (1 * 128) + (0 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (0 * 1) + (0 * 2) +$   
 $(1 * 1)$   
 110 100 001

1 的补值:001 011 110

2 的补值:001 011 111

检查原来的二进制数 + 它的 2 的补值, 结果为:

```

110 100 001
001 011 111
-----
000 000 000

```

C.19 0

### 练习题

- C.20 有些人认为采用十二进制数值系统, 会使所有计算变得简单。这是由于 12 与 10 (代表基数 10) 相比, 可被更多的数字除尽 (12 可被 1, 2, 3, 4, 6, 12 等整除; 而 10 只能被 1, 2, 5, 10 等整除)。请问, 在基数为 12 的数值系统中, 最小的数位是什么? 基数为 12 采用的最大数位号可能是什么? 对于采用基数 12 表达的任何数字来说, 最右侧 4 个数位的位值各为多少?
- C.21 在任何数值系统中, 对最右边那个数位左边第一个数位 (从右数第二个数位) 来说, 它的位值同数值系统的最大符号值之间存在什么关系?
- C.22 请补全下列指定数值系统的右侧 4 个数位的位值:

|       |       |     |     |     |
|-------|-------|-----|-----|-----|
| 十进制   | 1 000 | 100 | 10  | 1   |
| 基数 6  | ...   | ... | 6   | ... |
| 基数 13 | ...   | 169 | ... | ... |
| 基数 3  | 27    | ... | ... | ... |

- C.23 把二进制数 100101111010 转换为八进制和十六进制。
- C.24 把十六进制数 3A7D 转换为二进制。
- C.25 把十六进制数 765F 转换为八进制 (提示: 先把 765F 转换为二进制, 再转换为八进制)。
- C.26 把二进制数 1011110 转换为十进制。
- C.27 把八进制数 426 转换为十进制。
- C.28 把十六进制数 FFFF 转换为十进制。
- C.29 把十进制数 299 分别转换为二进制、八进制和十六进制。
- C.30 列出十进制数 779 的二进制形式, 再列出它的 1 的补值计算结果, 最后列出 2 的补值结果。
- C.31 一个数加上其 2 的补值, 会得到什么结果?
- C.32 用支持 32 位整数的机器, 整数值显示 -1 的 2 的补值运算结果。

## 附录 D 因特网和万维网上的C++ 资源

本附录列出了因特网和万维网上一些有价值的C++ 资源。这些资源包括 FAQ(常见问题解答)、教程、获取 ANSI/ISO C++ 正式标准的方式、常用C++ 编译工具的有关情况以及获取免费编译器、演示程序、书籍、教程、软件工具、文章、访谈录、会议、报刊杂志、在线课程、新闻组和求职等资源的方式。

欲详细了解美国国家标准协会(ANSI)或购买标准文档,请访问 ANSI 主页:  
<http://www.ansi.org/>。

### D.1 资源

<http://www.progsources.com/index.html>

这是一个涵盖多种程序语言(包括C++)的出色信息资源。可在此找到一系列工具、编译器、软件、书籍和其他C++ 资源。

<http://www.intranet.ca/~sshah/booklist.html#C++>

这里有一个C++ 专题书库,其中列出了 30 多本书。

<http://www.genitor.com/resource.htm>

非常出色的一个网站,通过该网站上的链接,您可以获得C++ 编译器、非常有用的C++ 工具、节选自其他 C/C++ 用户期刊和出版物的源代码。

<http://www.possibility.com/Cpp/CppCodingStandard.html>

这是一个内容相当丰富的网站,其中不仅包含了C++ 编程语言的详细论述,还包括了许多非常出色的C++ 源代码清单。

<http://help-site.com/cpp.html>

该网站提供了指向网上C++ 资源的链接。

<http://glenmcl.com/tutor.htm>

对已具备 C/C++ 初步知识的读者来说,该网站是非常优秀的一个资源库。各主题都有详细的解释和示范代码。

<http://programmersheaven.com/zone3/cat353/index.htm>

该网站收集了非常多的C++ 库,供用户免费下载。

<http://www.prorammerwsheaven.com/zone3/cat155/index.htm>

该网站提供 C/C++ 工具和库。

<http://www.programmersheaven.com/wwwboard/board3/wwwboard.asp>

该信息板允许用户在此张贴 C/C++ 编程方面的问题和评论。

<http://hal9k.com/cug/>

该网站提供C++ 资源、期刊、共享软件和免费软件等。

<http://developer.earthweb.cpm/directories/pages/dir.c.developmenttools.html>

颇受程序员欢迎的一个网站,它为 C 和C++ 程序员提供了涉及范围较广的资源列表。

<http://www.devx.com>

对程序员来说,devx 是一个非常全面的参考网站。通过这个网站,可了解到各种编程语言最新动态、工具和编程技巧。该网站的C++ 区专门提供和C++ 相关的内容。

## D.2 教程

<http://info.desy.de/gna/html/cc/index.html>

《Introduction to Object - Oriented Programming Using C++》(利用C++ 进行面向对象编程入门)教程目前已经可以在此下载,您也可以在线注册网上课堂。选择和面向对象编程及C++ 编程语言有关的参考书籍。

<http://uu-gna.mit.edu:8001/uu-gna/text/cc/Tutorial/tutorial.html>

《Introduction to Object - Oriented Programming Using C++》共有 10 章,每章都有一系列练习题及练习题答案。

<http://www.icce.rug.nl/docs/cplusplus/cplusplus.html>

该教程是一名大学教授专门针对打算学习C++ 的 C 程序员编写的。

<http://www.rdw.tec.mn.us/>

Red Wing/Winona 理工学院提供的C++ 网上课程。

<http://www.zdu.com/zdu/catalog/programming.htm>

ZD Net 网络大学提供各式各样与C++ 编程语言相关的在线课程。

<http://library.advanced.org/3074/>

该教程为打算学习C++ 的 Pascal 程序员设计。

<http://rtfm.mit.edu/pub/usenet/news.answers/C-faq/learn-c-cpp-today>

该网站提供了一个C++ 教程清单,此外还包含与各种C++ 编译器相关的信息。

<http://www.icce.rug.nl/docs/cplusplus/cplusplus.html>

该网站为了解 C 并打算学习C++ 的用户而设计。

<http://www.cprogramming.com/tutorial.html>

该网站有一个包含示范代码的入门教程。

<http://programmersheaven.com/zone3/cat34/index.htm>

该网站包含一个教程主题清单。它涉及范围广,不管初学者,还是专家,都能在此找到适合的主题。

## D.3 FAQ

<http://reality.sgi.com/austern/std-c++/faq.html>

这是一个 FAQ 站点,内容涉及C++ ANSI/ISO 标准、C++ 编程语言设计和C++ 语言的最新动态。

<http://www.trmpthrst.demon.co.uk/caplibsl.html>

这是一个C++ 库 FAQ 站点。您可以在这里找到关于C++ 标准库的常见问题及其解答。

<http://pneuma.phys.ualberta.ca/~burris/cpp.htm>

“Internet Link Exchange”是另一个非常优秀的C++ 信息资源。该网站提供指向 comp.lang.C++ 和C++ 标准库相关问题的链接。

<http://www.math.uio.no/nett/faq/C-faq/faq.html>

常见问题及其解答 comp.lang.c 列表。

[http://lgblwww.epfl.ch/~wolf/c\\_std.html](http://lgblwww.epfl.ch/~wolf/c_std.html)

有关 C 编程语言 ANSI/ISO 标准的常见问题列表。

<http://www.cerfnet.com/~mpcline/C++-FAQs-Lite/>

该网站的常见问题非常丰富,多达 35 类。

<http://faqs.org/faqs/by-newsgroup/comp/comp.lang.C++.html>

该网站有一系列常见问题和教程链接,内容来自于 Comp.Lang.C++ 新闻组。

<http://www.cerfnet.com/~mpcline/C++-FAQs-Lite/>

该网站涉及的内容非常广泛。每个主题都包括许多问题及其解答。

<http://www.eskimo.com/~scs/C-faq/top.html>

这里列出的常见问题中,内容涉及指针、内存分配和字符串。

## D.4 Visual C++

[http://chesworth.com/pv/languages/c/visual\\_cpp\\_tutorial.htm](http://chesworth.com/pv/languages/c/visual_cpp_tutorial.htm)

对初次接触 Microsoft Visual C++ 的人来说,这是个不可不看的好教程。该教程对C++ 作了一个简明扼要的介绍。

## D.5 comp.lang.C++

<http://weblab.research.att.com/phoaks/comp/lang/C++/resources0.html>

这里有关于 comp.lang.C++ 的非常丰富的信息资源。标题为《People Helping One Another Know Stuff》的页面总结了该网站涉及的内容。此外,通过这里的链接,还可以到达另外 40 个C++ 资源。

<http://www.r2m.com/windev/cpp-compiler.html>

该网站包含许多指向其他C++ 相关网站的链接。

<http://home.istar.ca/~stepanv/>

该网站有许多链接,通过这些链接,可以访问有关C++ 编程的许多文章和参考资源。该网站列出的主题包括面向对象的图像、ANSI C++ 标准、标准模板库、MFC 资源和教程。

<http://kom.net/~dbrick/newspage/comp.lang.C++.html>

通过该网站,您可以连接到与 comp.lang.C++ 相关的新闻组。

<http://www.austinlinks.com/CPlusPlus/>

Quadrabay 公司的网站。其中包括一些指向C++ 参考资源的链接。通过这些链接,您可以了解 Visual C++/MFC 库、C++ 编程信息、C++ 求职资源和一系列帮助您学习C++ 的教程及其他在线工具。

[http://db.csie.ncu.edu.tw/~kant\\_c/C/chapter2\\_21.html](http://db.csie.ncu.edu.tw/~kant_c/C/chapter2_21.html)

该网站有一个 ANSI C 标准库函数列表。

[http://wwwcn1.cern.ch/asd/geant/geant4\\_public/coding\\_standards/coding/coding\\_2.html](http://wwwcn1.cern.ch/asd/geant/geant4_public/coding_standards/coding/coding_2.html)

一个非常优秀的、内容丰富的C++ 标准信息参考资源网站。

<http://cuiwww.unige.ch/OSG/Vitek/Compilers/Year86/msg00046.html>

“分段式计算机的 C 标准”。

<http://www.csci.csusb.edu/dick/C++std/>

该网站提供了指向 ANSI/ISO C++ 标准草案的和 Usenet 新闻组 comp.std.C++ (提供该标准最新动态)的链接。

<http://ibd.ar.com/ger/comp.lang.C++.html>

Green Eggs Report 列出了上百个 comp.lang.C++ 内包含的 URL。

<http://www.ts.unu.se/~maxell/C++/>

该网站提供了一些示范代码,供部分C++ 类使用。

<http://www.quadrabay.com/CPlusPlus/>

这是一个相当出色的网站,内容包括C++ 编程信息、C++ 进阶、C++ 求职指导和其他与C++ 有关的主题。

<http://www.research.att.com/~bs/homepage.html>

这是 Bjarne Stroustrup (C++ 语言的设计者)的主页。他在此提供了C++ 参考资源、常见问题解答以及其他有价值的C++ 信息。

<http://cygnus.com/misc/wp/draft/index.html>

该网站有“进展中的 ANSI C++ 标准草案(HTML 格式的)”(1996 年 12 月)。

<http://www.austinlinkes.com/CPlusPlus/>

该网站列出了一些C++ 参考资源,内容包括推荐书目、求职信息、C++ 编程语言有关的信息和一些链接(通过这些链接,可以访问其中包括C++ 参考资源的其他网站)。

<ftp://research.att.com/dist/C++std/WP/CD2/>

该网站有当前的 ANSI/ISO C++ 草案标准。

<http://ai.kaist.ac.kr/~ymkim/Program/C++.html>

该网站提供教程、库、常见编译器、常见问题解答和新闻组。

<http://www.cyberdiem.com/vin/learn.html>

《Learn C/C++ Today》是该网站的主题,提供了许多关于 C/C++ 学习的中、高级教程。

<http://www.trumphurst.com/cplusplus1.html>

C++ 库常见问题集是编程高手编写的,其目的是供其他C++ 程序员使用。这里的内容是定期更新的,可从这里了解C++ 最新动态。

<http://www.experts-exchange.com/comp/lang/cplusplus/>

Experts Exchange 是一个免费的参考资源,高级技术专家希望借此与志同道合者分享信



息的网站。注册为会员之后,就可以张贴问题或回答问题。

<http://www.execpc.com/~ht/vc.htm>

该网站提供了指向C++ 编程语言的链接,通过这些链接,您可以访问常规信息网站、教程、杂志和图书馆。

<http://cplus.about.com/~ht/vc.htm>

这是 about.com 的 C/C++ 编程语言专区。可在此找到教程、免费/共享软件、辞典、求职信息、杂志和其他相关内容。

[http://pent21.infosys.tuwien.ac.at/cetus/oo\\_c\\_plus\\_plus.html#oo\\_c\\_plus\\_plus\\_general\\_newsgroups](http://pent21.infosys.tuwien.ac.at/cetus/oo_c_plus_plus.html#oo_c_plus_plus_general_newsgroups)

您可以在这里找到C++ 概述。该网站还提供了新闻组服务。

<news:comp.lang.C++>

这是专门解决面向对象C++ 语言问题的新闻组。

<news:comp.lang.C++.moderated>

针对专业人士的C++ 新闻组。

## D.6 编译工具

<http://www.progsources.com/index.html>

The Programmer's Source(程序员资源)是关于许多程序语言(包括C++)的一个出色信息资源。您将在此找到一系列工具、编译器、软件、书籍和其他C++ 资源。

<http://www.cygnus.com/misc/gnu-win32/>

Cygnus 网站的 GNU 开发环境可在此免费下载。

<http://www.remcomp.com/lcc-win32/>

可以在该网站免费下载用于 Windows95/NT 平台的 LCC - Win32 编译器。

<http://www.microsoft.com/visualc/>

Microsoft Visual C++ 主页提供了与 Visual C++ 有关的产品信息、概述、补充材料和订购信息。

<http://www.powersoft.com/products/languages/watccpl.html>

针对 WatcomC/C++ 11.0 版的 Powersoft 产品新闻和信息。该编译器不能通过网站下载,但这里提供了相关的购买信息。

<http://netserv.borland.com/borlandcpp/cppcomp/turbocpp.html>

Borland Turbo C++ Visual Edition for Windows 编译器网站。

[http://www.symantec.com/scpp/fs\\_scpp/fs\\_scpp72\\_95.html](http://www.symantec.com/scpp/fs_scpp/fs_scpp72_95.html)

Symantec C++ 7.5 for Windows 95 和 Windows NT。

<http://www.metrowerks.com/products/>

Metrowerks CodeWarrior for Macintosh 或 Windows。

<http://www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html>

该网站收集了来自 comp.compilers 新闻组的常见问题集。

<http://www.ncf.carleton.ca/%7Ebg283/>

这是用于 DOS 的 C++ 编译器,名为 Miracle C 编译器。该编译器可以免费下载,但其源代码只有在注册付费之后才能使用。

<http://www.borland.com/bcppbuilder/>

这是一个指向 Borland C++ Builder 5.5 的链接。通过该链接,您可以免费下载其命令行版本。

<http://www.compliers.net/>

Compliers.net 是一个网站,其目的是帮助您搜索编译器。

<http://sunset.backbone.olemiss.edu/%7Ebobbcook/eC/>

这是一个C++ 编译器,是为那些打算从 Pascal 转而使用C++ 的C++ 初级用户设计的。

<http://developer.intel.com/vtune/compliers/cpp/>

Intel C++ 编译器。能支持该编译器的平台有 Windows 98、NT 和 2000。

[http://www.kai.com/C\\_plus\\_plus/index.html](http://www.kai.com/C_plus_plus/index.html)

Kai C++ 编译器有 30 天免费试用期。

## D.7 开发工具

<http://www.quintessoft.com/>

Quintessoft Engineering 公司提供C++ 代码导航器,该导航器是用于 Windows95/NT 的 C++ 开发工具。您可以在这里找到关于该开发工具的产品信息、用户意见、免费试用版下载和购买信息。

## D.8 标准模板库

### 教程

<http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html>

该 STL 教程按照示例、体系、组件和扩展性 STL 进行组织的。利用 STL 组件、有价值的解释性内容和非常有用的图示,您可以找到示范代码。

[http://web.ftech.net/~honeyg/articles/eff\\_stl.htm](http://web.ftech.net/~honeyg/articles/eff_stl.htm)

该 STL 教程提供的信息非常丰富,内容涉及 STL 组件、容器、流和迭代器适配器、变换和选定数值、筛选和变换数值,以及对象等。

[http://www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

该网站适合才接触 STL 的读者。可在此迅速找到 STL 和 ObjectSpace STL Tool Kit 示例。

### 参考资源

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

该网站列举了许多 STL 网站,还推荐了一些有助于进一步了解 STL 的参考书目。

<http://www.cs.rpi.edu/projects/STL/stl/stl.html>

Standard Template Library Online Reference Home Page(标准模板库网上参考资源)是 Rensselaer(伦斯莱尔)理工大学设立的主页。你可以在此找到对 STL 的详细解释以及指向其他 STL 相关资源的链接。

<http://www.sgi.com/Technology/STL>

Silicon Graphics Standard Template Library Programmer's Guide(硅谷图形标准库程序员指南)是一个非常有用的 STL 参考资源。你可以从该站点下载 STL,查找最新信息、设计文档以及指向其他 STL 网上资源的链接。

<http://www.dinkumware.com/refcpp.html>

该网站包含一些相当有价值的 ANSI/ISO 标准C++ 库相关信息,此外还包含标准模板库的更多信息。

<http://www.roguewave.com/products/xplatform/stblib/>

Rogue Wave Software's Standard C++ Library web page (Rogue Wave 软件公司的标准 C++ 库网页)。你可以在此下载其与标准C++ 库版本相关的白皮书。

### 常见问题解答

<http://butler.hpl.hp.com/stl/stl.faq>

该 FTP 站点是 Marian Corcoran 维护的 STL 常见问题解答集,她是 ANSI 委员会的成员之一,同时也是一名资深的C++ 专家。

### 论文、书籍和访谈

[http://www.sgi.com/Technology/STL/other\\_resources.html](http://www.sgi.com/Technology/STL/other_resources.html)

该网站列举了许多 STL 相关网站,而且还推荐了部分优秀的 STL 参考书。

<http://www.byte.com/ate/9510/sec12/arr3.htm>

《Byte Magazine》有一篇 Alexander Stepanov 撰写的文章。作为标准模板库的创始人之一,Stepanov 慷慨公开他所积累的 STL 使用技巧,与读者交流。

<http://www.sgi.com/Technology/STL/drdobbs-interview.html>

Alexander Stepanov 访谈录,其中包括创建标准模板库的一些趣闻轶事。Stepanov 讲述了 STL 概念的由来,普通的编程技巧、STL 的全称等。

### ANSI/ISO C++ 标准

<http://www.ansi.org/>

可从该网站购买C++ 标准文档的副本。

### 软件

<http://www.cs.rpi.edu/~musser/stl.html>

RPI STL 网站包含下列相关信息:STL 和其他C++ 库相比,究竟存在哪些不同;如何编译

使用了 STL 的程序,其中包含文件的主要 STL 列表、使用了 STL、STL 容器类和 STL 迭代其类别的演示程序。与此同时,该网站还提供了一个兼容 STL 的编译器列表,提供了包含 STL 源代码和相关参考材料的 FTP 站点。

<http://www.mathcs.sjsu.edu/faculty/horstman/safestl.html>

下载 SAFESTL.ZIP,要查找采用了 STL 的程序中的错误,就可以利用该工具。

<http://www.objectspace.com/jgl/>

Object Space(对象空间)提供把C++ 移植到 Java 的相关信息。您可以在这里免费下载他们的 Standards <ToolKit> 可移植类库。该工具包的重要特性包括容器、迭代器、算法、分配符、字符串和异常等。

<http://www.cs.rpi.edu/~wiseb/stl--borland.html>

Using the Standard Template Library with Borland C++ (Borland C++ 标准模板库使用指南)是一个非常出色的网站,对于使用 Borland C++ 编译器的用户来说,尤其如此。作者对相关注意事项和不兼容等问题进行了深入的描述。

<http://msdn.microsoft.com/visualc/>

这是 Microsoft Visual C++ 的主页。您可以在这里了解到 Visual C++ 的最新动态、更新、技术参考资源、演示程序并下载最新工具等。

<http://www.borland.com/bcppbuilder/>

这是 Borland C++ Builder 的主页。您可以在这里找到大量的C++ 参考资源,其中包括 C++ 新闻组、最新产品相关信息、常见问题解答以及其他参考资源。对使用C++ Builder 的程序员来说,这是一个不可不去的优秀网站。